27 独自のプログラムを設計したい！

27.1 API 関数を使ってやりたいこと

■ 1 局の PLC のデバイスをリード／ライトしたい

シングルハンドル系関数（☞27.1.1 シングル／マルチハンドル系関数について）

- リード系 API
  - キャッシュタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
  - ダイレクトタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
- ライト系 API
  - ダイレクトタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
  - 書き込み後キャッシュリフレッシュ付き（☞27.1.2 キャッシュ／ダイレクトタイプについて）

■ 複数の PLC 通信をしたい

マルチハンドル系関数（☞27.1.1 シングル／マルチハンドル系関数について）

- リード系 API
  - キャッシュタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
  - ダイレクトタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
- ライト系 API
  - ダイレクトタイプ（☞27.1.2 キャッシュ／ダイレクトタイプについて）
  - 書き込み後キャッシュリフレッシュ付き（☞27.1.2 キャッシュ／ダイレクトタイプについて）

■ 効率よく通信させたい

・グループシンボルアクセス（☞27.1.4 グループアクセスについて）
・キューイングアクセス（☞27.1.5 キューイングアクセスについて）

■ その他の関数

・システム系 API（☞27.1.7 システム系 API について）
・SRAM 内データアクセス API（☞27.1.8 SRAM 内データアクセス API について）
・CF カード関係 API（☞27.1.9 CF カード関係 API について）
・その他 API（☞27.8 その他の API）

27.1.1 シングル／マルチハンドル系関数について

シングルハンドル系

シーケンシャルに相手と通信する API で、ある API を呼び出している間は、別の API を呼び出すことはできません。

そのかわり、API を呼び出すときに、『Pro-Server EX』のアクセスハンドルの取得などの面倒な手続きなしに API を呼び出すことができます。

マルチハンドル系

シングルハンドル系 API の機能を、複数の相手に対して同時に使用することを可能にした API です。

マルチハンドル系 API では、シングルハンドル系 API と区別するために、API 名の最後に大文字の「M」が付きます。

例えば、シングルハンドル系 API の「ReadDeviceVariant()」と同等の機能を持つマルチハンドル系 API は、「ReadDeviceVariantM()」となります。

マルチハンドル系 API は、マルチスレッドを利用する場合や、複数の接続機器に同時にアクセスする場合などに使用します。

27.1.2 キャッシュ／ダイレクトタイプについて

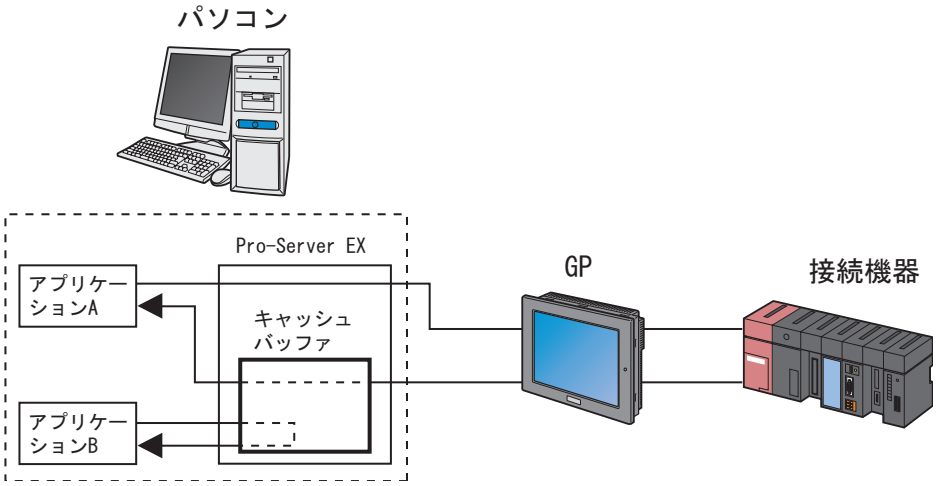
キャッシュリード

複数のアプリケーションが同じデバイスに対し読み出し（リード）するとき、『Pro-Server EX』がそれらの要求に対し、一つ一つ、接続機器に対し読み出しを行うと時間が掛かります。

キャッシュリードでは、アプリケーション A、B の 2 つが、同じ接続機器の同じデバイスに対しリード要求した場合、まずアプリケーション A の要求に対し接続機器からデータを読み出し、それを『Pro-Server EX』内のキャッシュバッファに蓄積し、かつアプリケーション A に読み出しの答えとして応答します。

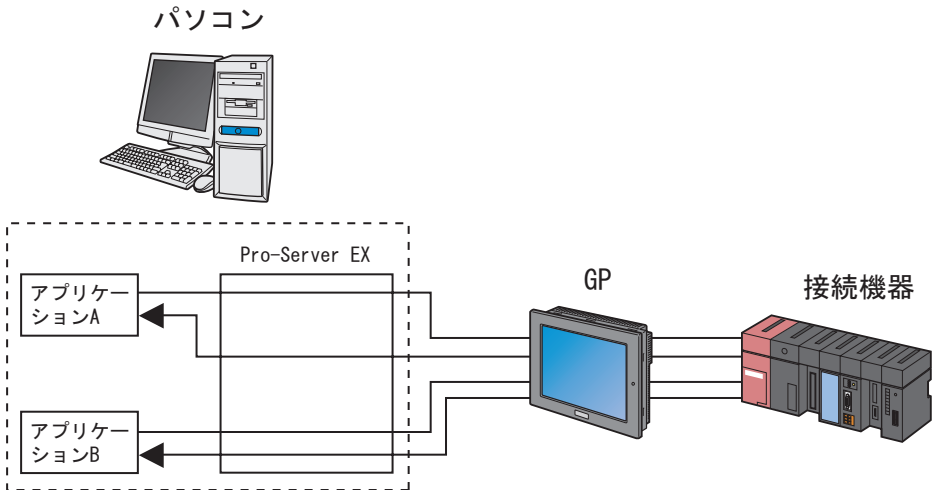
次に、アプリケーション B からのリード要求には、アプリケーション A の時に読み出したデータがすでにキャッシュバッファに存在しますので、それを返します。

『Pro-Server EX』には、キャッシュバッファ制御用の API も用意されています。詳細については、「27.3 キャッシュバッファ制御 API」を参照してください。



ダイレクトリード

キャッシュ状況に関係なく、常に接続機器から最新のデータを読み出します。
ダイレクトリードの場合は、API 名の最後に大文字の「D」または「DM」が付きます。



ダイレクトライト

値を書き込む API です。ダイレクトライトの場合は、API 名の最後に大文字の「D」または「DM」が付きます。

キャッシュリフレッシュ付きライト

あるデバイスをキャッシュしている場合に、値を書き込んだあと、その値でキャッシュ値を更新します。ダイレクトライトに比べると処理速度は劣ります。
デバイスを『Pro-Server EX』でキャッシュリードしているときは、キャッシュリフレッシュ付きライト API を使用してください。

27.1.3 キャッシュバッファ制御について

キャッシュバッファ制御 API は、キャッシュ対象のデバイスについて、キャッシュデータが更新されているかどうかを知るための API です。

MEMO
・ キャッシュバッファ制御 API は、ネットワークプロジェクトファイルを書き換えるための API ではなく、『Pro-Server EX』内のメモリへの情報の追加および変更を行います。

■ キャッシュバッファについて

『Pro-Server EX』では、デバイスデータをキャッシュするとき、複数のデバイスをまとめて管理していますが、その管理単位をキャッシュバッファと呼びます。

① 1 キャッシュバッファは、複数のレコードで構成されます。
② 1 レコードは、連続する複数のデバイスをデバイスアドレスで直接指定するか、シンボルによる指定、もしくはグループシンボルによる指定が可能です。
③キャッシュバッファは、キャッシュバッファ単位で独自の名前を付けることができます。

MEMO
・ キャッシュバッファを登録する方法については、以下の 2 つがあります。
①『Pro-Studio EX』で登録（機能画面の「デバイスキャッシュ」で作成しネットワークプロジェクトファイルに登録）

☞「29.5 **よく使用するデバイスのキャッシュ登録**」

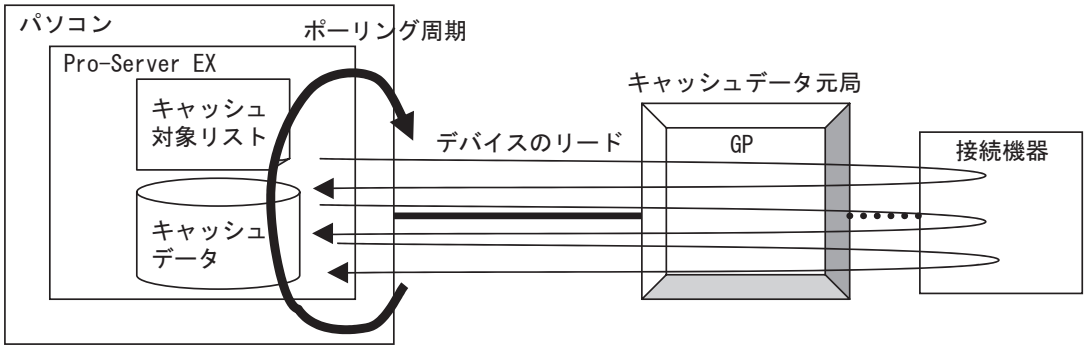
② API を利用した登録

■ キャッシュの更新方法

キャッシュバッファの更新方法には、“ ポーリング方式 ” と “ 常時監視方式 ” があります。

◆ポーリング方式のしくみ

キャッシュバッファ登録時に指定された周期になると、『Pro-Server EX』がキャッシュバッファ内のキャッシュ対象のリストに従い、デバイスをリードしてキャッシュバッファを更新します。

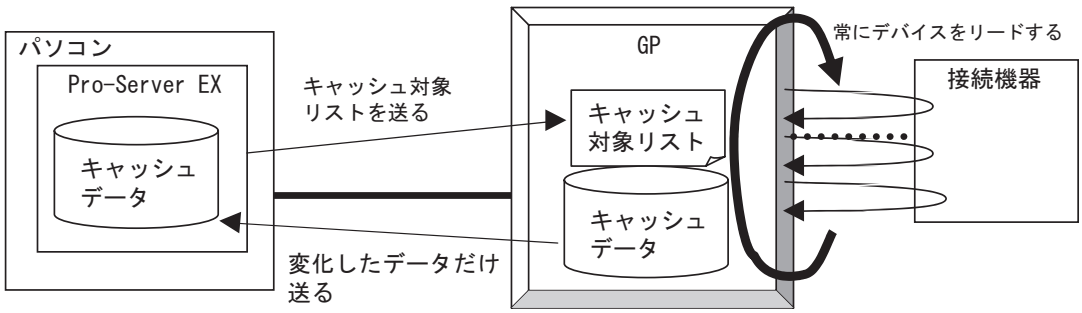


◆常時監視方式のしくみ

『Pro-Server EX』はキャッシュバッファの更新を開始すると、データ元局にキャッシュ対象のリストを送ります。

データ元局はそのリストに従い、常に（できる限り高速に）データをリードし、変化のあったものだけを『Pro-Server EX』に送ります。

『Pro-Server EX』はそれを受け取り、キャッシュデータとします。



MEMO

- キャッシュデータのデータ元となる局が GP シリーズ局の場合、常時監視方式は利用できません。

■ 常時監視方式とポーリング方式の使い分け

大量のデバイスを常時監視方式で監視すると、そればかりに処理が追われ、システム全体のパフォーマンスが低下します。
そのための対策として、緊急性の高いものだけを常時監視方式にして、それ以外はポーリング方式にすることをおすすめします。
ポーリング方式の場合、パソコンやネットワークの状況、対象接続機器の種類など、ご使用のシステムのパフォーマンスによっては、更新周期通りに更新しないことがあります。この場合はダイレクトリードをご使用ください。
データ量と方式は、目安として 常時監視方式の場合は数十～数百バイトまで、ポーリング方式の場合は数 K バイトまで、それ以上はダイレクトリードをご使用ください。
但し、ご利用のシステムのパフォーマンスによってバイト数は違います。

■ キャッシュの開始と停止

『Pro-Server EX』のキャッシュ動作の開始 / 終了タイミングについて説明します。

①キャッシュはキャッシュバッファ単位で開始、停止します。

②『Pro-Studio EX』で、ネットワークプロジェクトファイルにキャッシュバッファを登録する場合、キャッシュバッファごとに以下の 3 通りの登録方法が指定できますが、それぞれの開始タイミングについて説明します。

1）『Pro-Server EX』起動時
『Pro-Server EX』が起動し、ネットワークプロジェクトがロードされたあと、開始します。
さらに、ネットワークプロジェクトがリロードされた場合も、その都度開始します。

2) 登録済みデバイスをリードしたとき、自動的に開始
キャッシュバッファ内に登録されているキャッシュデバイスに対しデバイスリードが発生した場合、開始します。
一部のデバイスに対してリードが行われても、キャッシュバッファ全体のキャッシュ動作が開始します。
開始の対象となるリード方法は、デバイスリード系の API でデバイスリードした場合以外でも（データ転送機能でデータ元となった場合や起動条件のチェック対象デバイスの場合など）、すべてのリード方法が対象です
この方法で開始された場合のみ、キャッシュバッファの対象デバイスへのアクセスが指定した時間以上無くなるとキャッシュ動作を停止します。

3）キャッシュバッファ開始 API（PS_StartCache）を使用し、プログラムから開始

③以下の場合、キャッシュ動作は停止します。

1）『Pro-Server EX』が終了したとき、キャッシュバッファは停止し、キャッシュデータを破棄します。

2）ネットワークプロジェクトをリロードする場合、リロードする直前にキャッシュバッファは停止し、キャッシュデータを破棄します。

3)「登録済みデバイスをリードした時、自動的に開始する」設定を有効にし、かつ、停止時間が設定されているキャッシュバッファのキャッシュ動作が開始され、設定されている時間以上アクセスがなかったとき停止します（破棄はされません）。
4）キャッシュ動作停止 API(PS_StopCache) を使用し、プログラムにて停止したとき

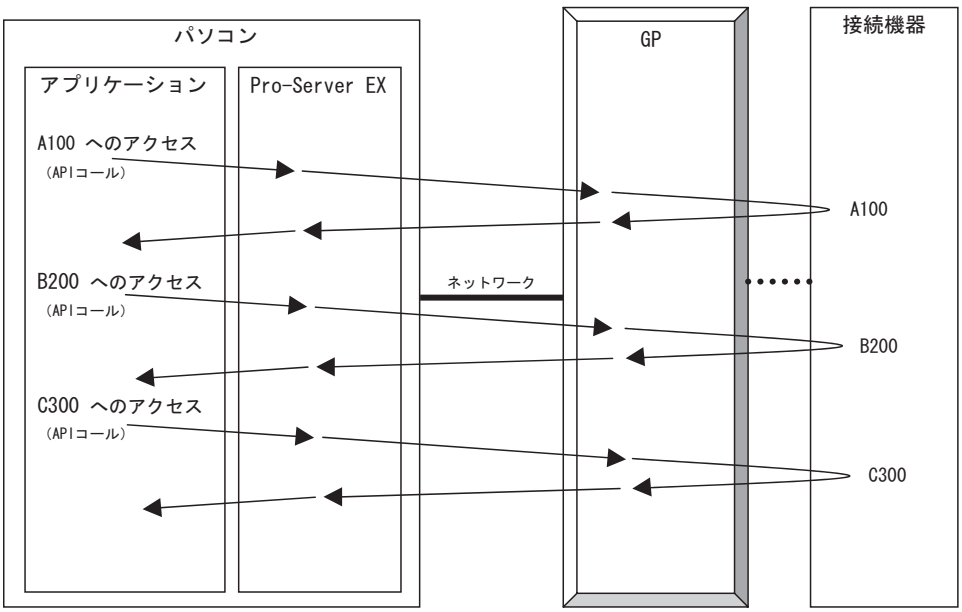
27.1.4 グループアクセスについて

デバイスアドレスを指定する場合に、グループシンボルを使用することができる API があります。
グループシンボルを使用すると、1 回の API コールで効率よく複数のデバイスにアクセスできます。

MEMO
- グループ内に複数のデバイスを持つグループシンボルを利用したアクセスの場合、高速にアクセスするため、『Pro-Server EX』や GP 内部で最適化処理が行われます。そのため、デバイスへのアクセスの順番は指定できません。（グループシンボル登録時のシンボルの登録順番でアクセスするわけではありません。）
  また、複数のデバイスのうち、1 つでもアクセスエラーが発生した場合、処理はそこで中断され、グループ全体のアクセスエラーとなり、残りのデバイスへのアクセスは実行されません。
- 1 回の API コールで利用可能なグループシンボルのデータサイズの最大は 1MByte 以内です。

◆各デバイスに対し、API を個別にコールした場合

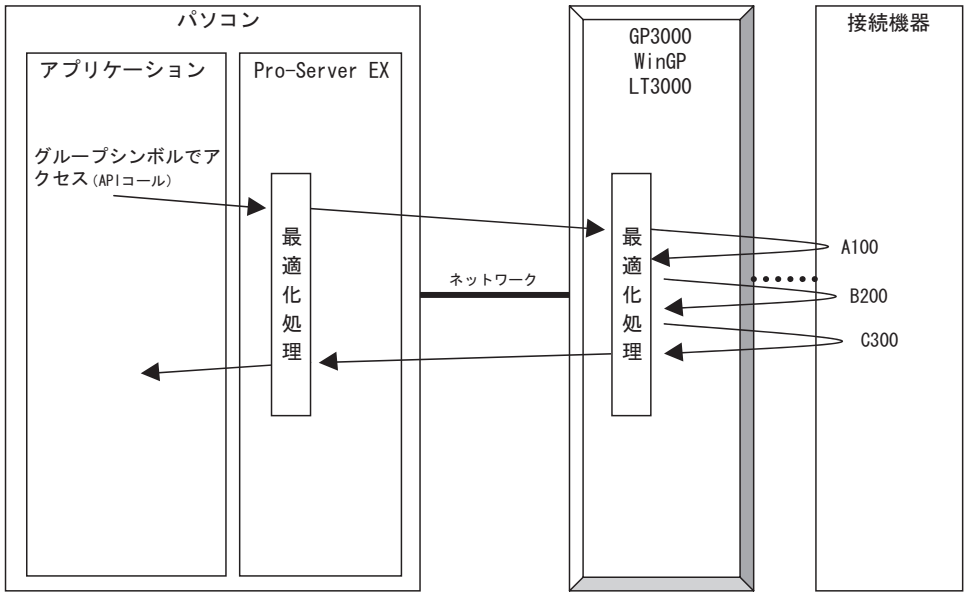
『Pro-Server EX』はそのつど、機器と通信します。



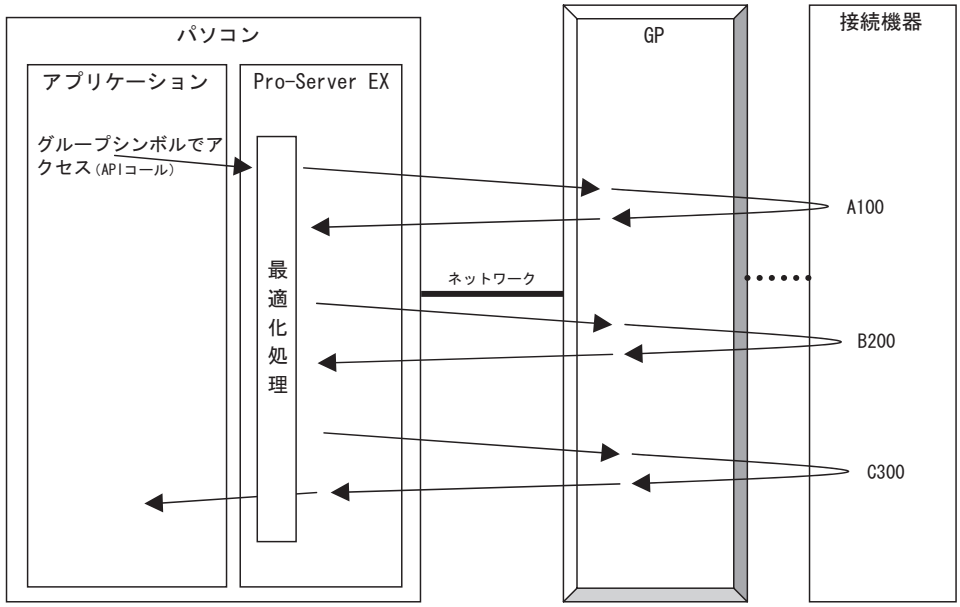
◆グループシンボルにアクセスする場合

動作は、GP3000 シリーズ局、WinGP 局および LT3000 局と GP シリーズ局で異なります。

- GP3000 シリーズ局、WinGP 局および LT3000 局の場合
  『Pro-Server EX』は GP3000 シリーズ局、WinGP 局および LT3000 局に対し、1 回の要求しか行いません。GP3000 シリーズ局、WinGP 局および LT3000 局は内部でその要求に対し接続機器へ分割アクセスをしますので、ネットワーク上効率よく通信が可能になります。



- GP シリーズ局の場合
  API コールは 1 回のみですが、『Pro-Server EX』内部で分割して GP シリーズ局にアクセスします。ただし、グループ内に複数のシンボルがあり、それらが連続している場合、『Pro-Server EX』はそれらを一括してアクセスします。



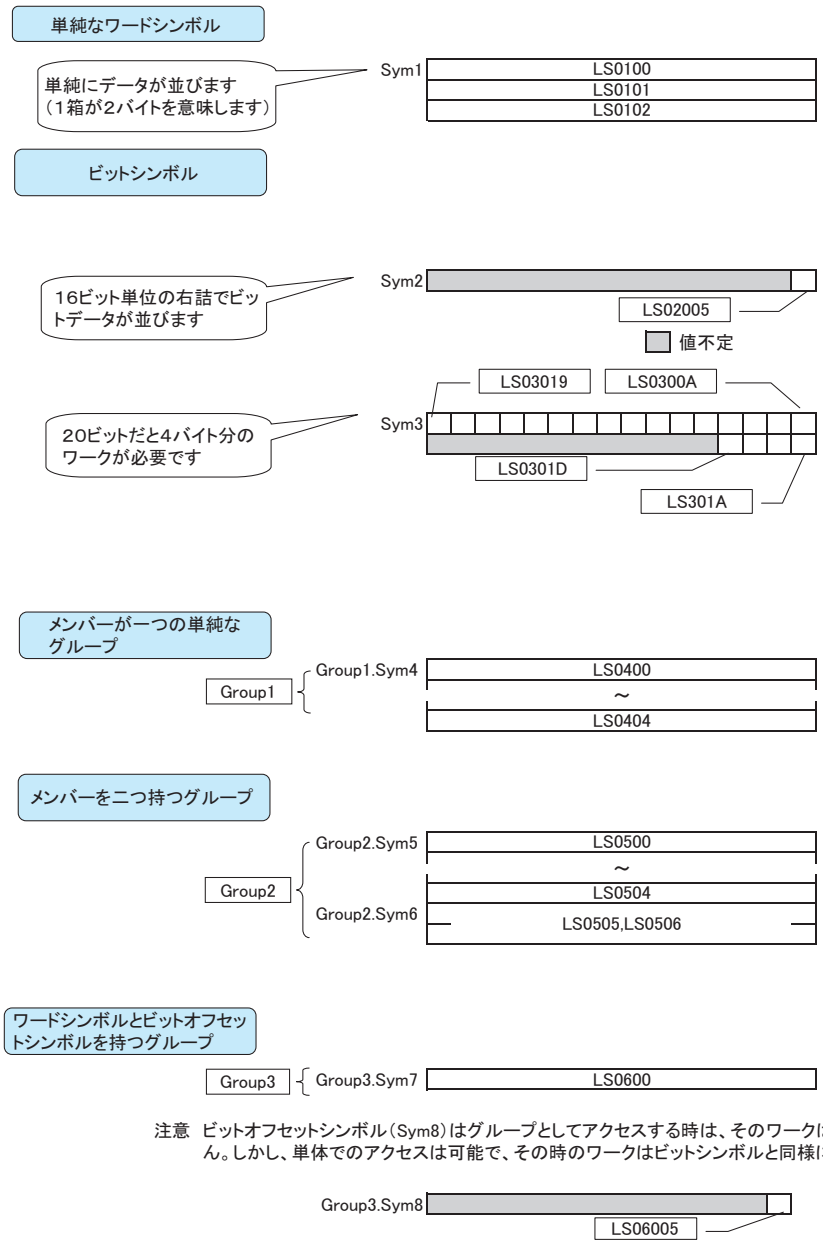
■ グループシンボルによるアクセス時のデータ構造

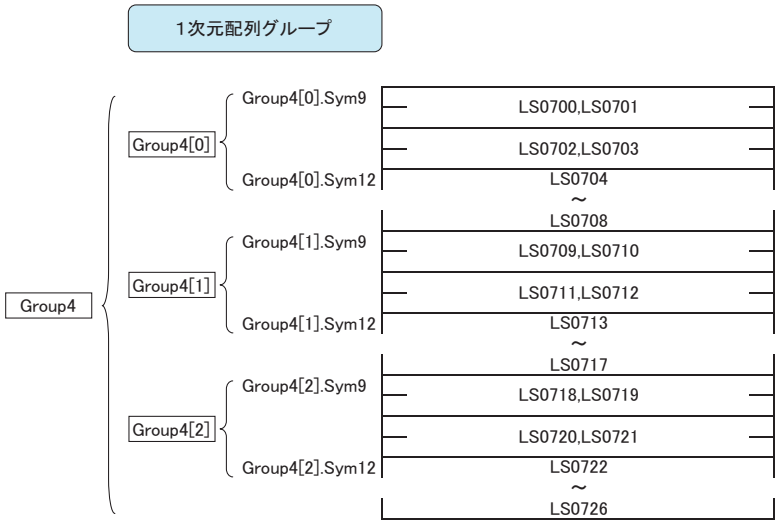
グループシンボルでデバイスにアクセスする場合のデータバッファ構造は、グループ内のシンボルの種類やサイズにより異なります。以下にその種類ごとのデータ構造を示します。

グループ内シンボルのデータタイプ	確保されるデータサイズ
Bit Data	• シンボルがビットシンボルの場合 16 ビット単位でデータ領域が確保されます。 • シンボルがビットオフセットシンボルの場合 データ領域は確保されません。
16Bit（Signed）Data	1 デバイス、2 バイト分のデータ領域が確保され、値はバイナリ値です。
16Bit（Unsigned）Data	
16Bit（HEX）Data	
16Bit（BCD）Data	1 デバイス、2 バイト分のデータ領域が確保され、実際のデバイスとアクセスするときに BCD 値←→バイナリ値変換が行われます。
32Bit（Signed）Data	1 デバイス、4 バイトのデータ領域が確保され、値はバイナリ値です。
32Bit（Unsigned）Data	
32Bit（HEX）Data	
32Bit（BCD）Data	1 デバイス、4 バイト分のデータ領域が確保され、実際のデバイスとアクセスするときに BCD 値←→バイナリ値変換が行われます。

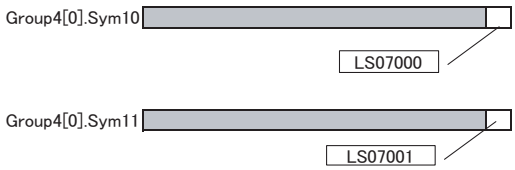
グループ内シンボルのデータタイプ	確保されるデータサイズ
単精度浮動小数点	1 デバイス、4 バイト分のデータ領域が確保され、値は単精度浮動小数点値として扱われます。
倍制度浮動小数点	1 デバイス、8 バイト分のデータ領域が確保され、値は単精度浮動小数点値として扱われます。
文字列データ	1 文字 1 バイトのデータ領域が確保されます。NULL 終端の文字列として扱われます。

データ構造のサンプルを以下に示します。

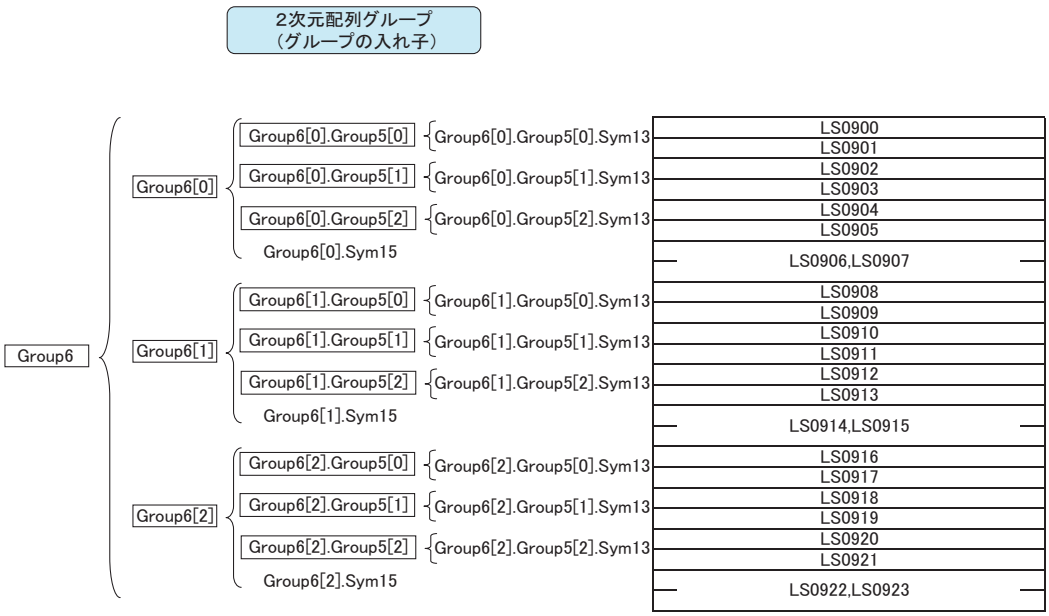




注意　ビットオフセットシンボル（Sym10,Sym11）はグループとしてアクセスする時、そのワークは存在しません。しかし、単体でのアクセスは可能で、その時のワークはビットシンボルと同様になります。

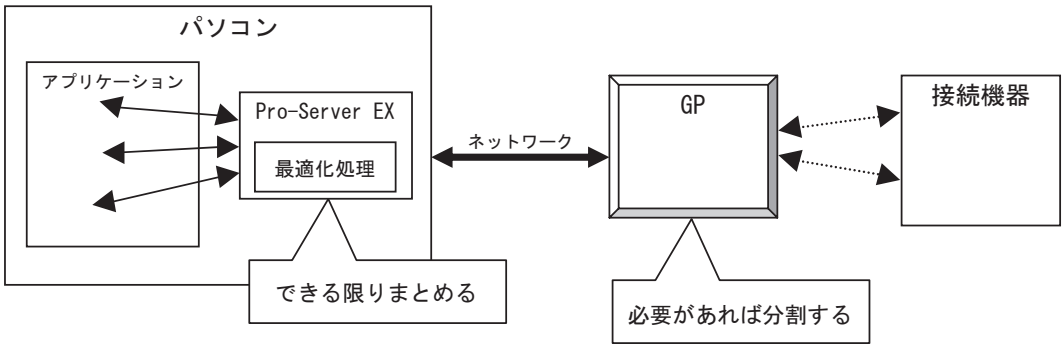


Group4[1].Sym10,Group4[1].Sym11のデバイスアドレスはそれぞれLS07090,LS07091になります
Group4[2].Sym10,Group4[2].Sym11のデバイスアドレスはそれぞれLS0718,LS07181になります



27.1.5 キューイングアクセスについて

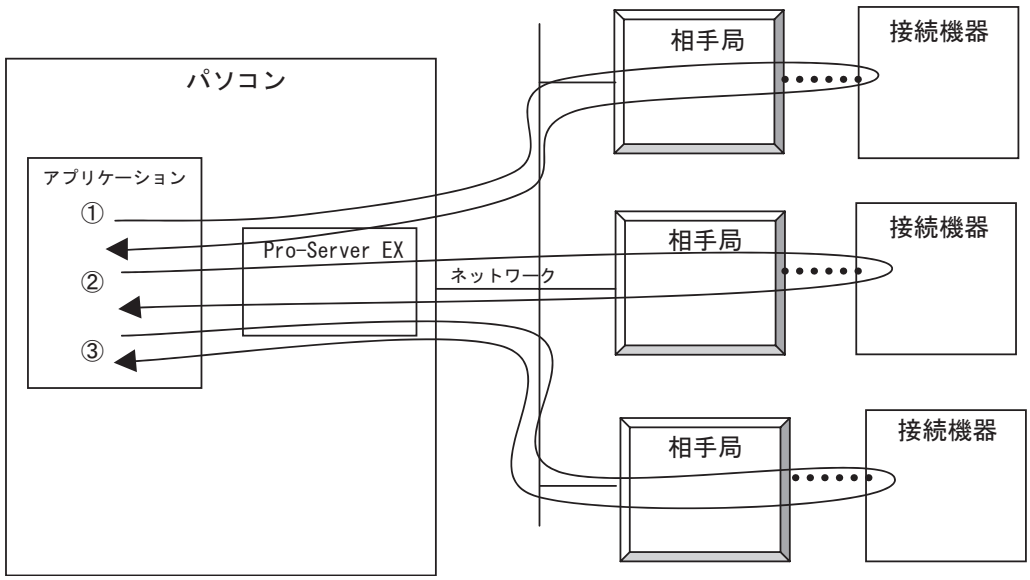
デバイスへのアクセスを API コールごとに蓄積し、その後、蓄積した要求を最適化し、まとめてアクセスする方法です。



キューイングアクセスのしくみ

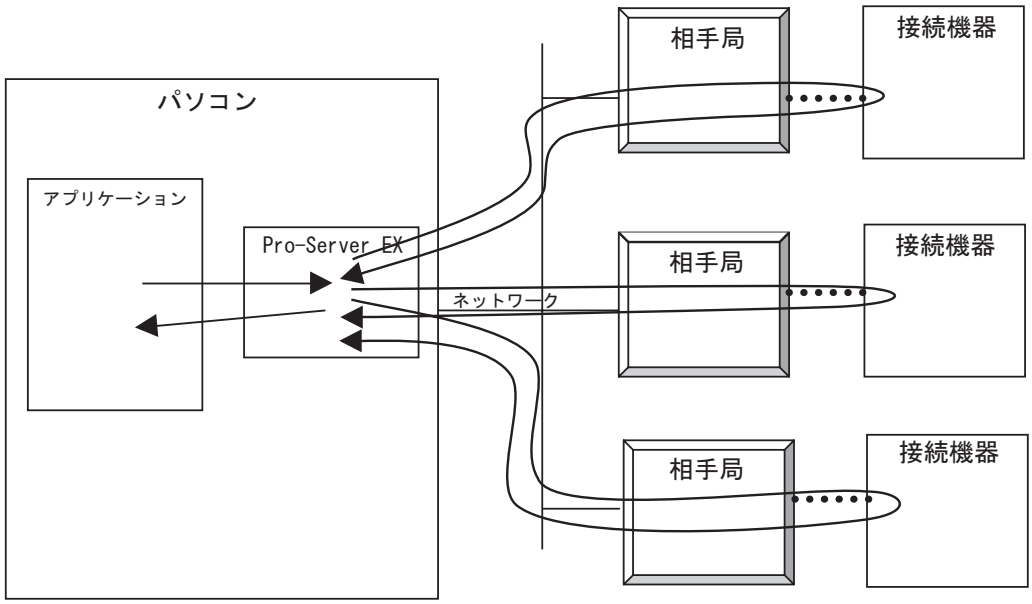
◆単純に API を利用したアクセス

処理はシーケンシャルに実行されます。



◆キューイングアクセス

相手局ごとに並列処理されます。



使用方法

①キューイングアクセスの開始を宣言します。（BeginQueuingRead() または BeginQueungWrite() をコールします。）

②デバイスリード系またはデバイスライト系の API をコールします。
（ReadDevice16() や WriteDevice16() などをコールします。）
このとき、引数に異常がなければ、API はすぐに復帰し、デバイスアクセスの要求だけが『Pro-Server EX』内に蓄積されます。この作業を“ アクセス要求の登録 ”と呼びます。

③蓄積されているデバイスアクセスの要求を実際に実行するため、ExecuteQueuingAccess() をコールします。『Pro-Server EX』はこの時点でデバイスアクセスの要求を最適化し、効率よく機器と通信を試みます。
ExecuteQueuingAccess() は、すべてのデバイスへのアクセスが成功すると成功を返し、一つでも失敗するとアクセスエラーを返します。
もし、デバイスごとにアクセスの成否が知りたい場合は、IsQueuingAcceessSucceeded() をコールし確認します。

MEMO

- キューイングアクセスを使用する場合、リードアクセスとライトアクセスの混在はできません。例えば、リードアクセス用としてキューイングアクセスの開始を宣言すると、ライトアクセスを登録することはできません。同様に、ライトアクセス用として開始すると、リードアクセスは登録できません。
  ただし、キューイングアクセスは、Pro-Server ハンドル単位で登録するので、別々の Pro-Server ハンドルでライトアクセス用、リードアクセス用と分けることは可能です。
- 一度アクセス要求の登録を行うと、同じデバイスへ同じ方法でアクセスする場合、再登録する必要はありません。
  『Pro-Server EX』は Pro-Server ハンドル単位でアクセス要求を記憶しているため、ExecuteQueuingAccess() がコールされるとその記憶を元に、何度でも実行します。
  アクセス要求の登録の記憶は、以下の場合にクリアされます。
  ①記憶している Pro-Server ハンドルが破棄されたとき
  ②新たなキューイングアクセスの登録が開始されたとき
  ③キューイングアクセスの登録をキャンセルしたとき（CancelQueuingAccess() をコールしたとき）
  また、ExecuteQueuingAccess() 実行後にエラーコードの文字列変換関数 (EasyLoadErrorMessage など) 以外を実行すると、それまでにキューイングされていたデータは破棄され、新たにキューイングが開始されます。
- 『Pro-Server EX』は、“ アクセス要求の登録 ” 時に、アクセスするためのデータバッファのアドレスを記憶します。（データではなく、アドレスだけを記憶します）。
  そのため、“ アクセス要求の登録 ” 後、ExecuteQueuingAccess() をコールして復帰してくるまで、データバッファは “ アクセス要求の登録 ” のアドレスに存在し続ける必要があります。
  そうしないと、『Pro-Server EX』は不正なアドレスにアクセスしてしまい、致命的エラーになることがあります。
  また、キューイングアクセスを再使用する場合も、その間、データバッファは “ アクセス要求の登録 ” のアドレスに存在し続ける必要があります。

27.1.6 ビットデータのアクセスについて

『Pro-Server EX』は、ビットデバイスへアクセスする場合、そのビットデータを扱う方法として 3 種類の方法を提供しています。

① 16 ビット単位での扱い：ビットデバイスに対し、16 ビット単位のビット列として扱う方法です。
ビットデータは D0 ビットから指定された個数分、右詰めで格納 / 使用されます。
データバッファは指定個数が 1 個でも、16 ビット分必要となります。また、指定個数に 16 ビット単位で必要になります。

（例）20 個のビットデバイスを指定した場合のデータバッファの格納順序

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

＜適用 API ＞

ReadDeviceBit/WriteDeviceBit(),

ReadDevice/WriteDevice(),ReadDeviceVariant/WriteDeviceVariant() で、データ型に 1（EASY_AppKind_Bit）を指定した場合

ReadSymbol/WriteSymbol() で、ビットシンボルやビットシンボルを含むグループを指定した場合

② Variant の BOOL 単位での扱い：1 ビットを Variant の BOOL データとして扱う方法です。
データバッファは 1 ビットが 1Variant の BOOL 型となり、指定個数分の BOOL 型の配列として扱います。

＜適用 API>

ReadDeviceVariant/WriteDeviceVariant() で、データ型に 0x201（EASY_AppKind_BOOL）を指定した場合

ReadSymbolVariant/WriteSymbolVariant() で、ビットシンボルやビットシンボルを含むグループを指定した場合

③グループシンボルによるアクセス時のビットオフセットシンボルの扱い
ビットオフセットシンボルを直接指定してデバイスへアクセスした場合、そのデータバッファは、上記で説明した “ 16 ビット単位での扱い ” か “ Variant の BOOL 単位での扱い ” になります。
ただし、グループシンボル内にビットオフセットシンボルがあり、そのグループシンボルでデバイスへアクセスした場合、データバッファ内にそのビットオフセットシンボル用のデータ領域は確保されません。
ビットオフセットシンボルはそのシンボルが単体で存在することはなく、必ず、その親となるワードシンボルがあります。データ領域としてはその親の分が確保されるので、ビットオフセットシンボルの分はその親の分の一部を利用してください。
詳細は、「27.1.4　グループアクセスについて」を参照してください。

27.1.7 システム系 API について

システム系 API は、『Pro-Server EX』の起動や終了、ネットワークプロジェクトファイルのロードなど、システム制御を行うための API です。
システム系 API には、以下の種類があります。

シングルハンドル系

Pro-Server ハンドルを指定せずに、『Pro-Server EX』の機能を使用する方法です。
この方法では、複数の API を同時に使用することはできません。（同時に API を使用すると、二重呼び出しのエラーになります。）

マルチハンドル系

Pro-Server ハンドルを指定して、『Pro-Server EX』の機能を使用する方法です。
異なる Pro-Server ハンドルを使用すると、API の同時利用が可能です。

27.1.8 SRAM 内データアクセス API について

GP に内蔵されている SRAM には、GP の設定や動作状況により、多種のデータが生成され格納されています。
その SRAM 内のデータにアクセスするための API です。
SRAM 内データアクセス API は、すべてシングルハンドル、マルチハンドルの両方をサポートしています。
ここではシングルハンドルで説明していますが、マルチハンドルの場合は API 名の最後に M が付き、第一引数に Pro-Server ハンドルが付加されます。

27.1.9 CF カード関係 API について

GP に内蔵されている CF カードには、SRAM と同様、GP の設定や動作状況によって多種のデータが生成され格納されています。
その CF カード内のデータにアクセスするための API です。

MEMO
- LT3000 シリーズおよび GP3200 には CF カードスロットがないため、CF 関係 API を使用することができません。

27.2 デバイスアクセス系 API

■ シングルハンドル系キャッシュリード API

関数名	ビットデータ
INT WINAPI ReadDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI ReadDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI ReadDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI ReadDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI ReadDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI ReadDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI ReadDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
関数名	文字列データ
INT WINAPI ReadDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI ReadDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI ReadDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI ReadSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI ReadSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ シングルハンドル系ダイレクトリード API

関数名	ビットデータ
INT WINAPI ReadDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI ReadDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI ReadDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI ReadDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI ReadDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI ReadDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI ReadDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
関数名	文字列データ
INT WINAPI ReadDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI ReadDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI ReadDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI ReadSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI ReadSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ シングルハンドル系ダイレクトライト API

関数名	ビットデータ
INT WINAPI WriteDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI WriteDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI WriteDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI WriteDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI WriteDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI WriteDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI WriteDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
関数名	文字列データ
INT WINAPI WriteDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI WriteDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI WriteDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI WriteSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI WriteSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ シングルハンドル系書き込み後キャッシュリフレッシュ付きライト API

関数名	ビットデータ
INT WINAPI WriteDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI WriteDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI WriteDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI WriteDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI WriteDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI WriteDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI WriteDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
関数名	文字列データ
INT WINAPI WriteDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI WriteDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI WriteDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI WriteSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI WriteSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ マルチハンドル系キャッシュリード API

関数名	ビットデータ
INT WINAPI ReadDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI ReadDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI ReadDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI ReadDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI ReadDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI ReadDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI ReadDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
関数名	文字列データ
INT WINAPI ReadDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI ReadDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI ReadDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI ReadSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI ReadSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ マルチハンドル系ダイレクトリード API

関数名	ビットデータ
INT WINAPI ReadDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI ReadDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI ReadDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI ReadDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI ReadDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI ReadDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI ReadDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
関数名	文字列データ
INT WINAPI ReadDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI ReadDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI ReadDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI ReadSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI ReadSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ マルチハンドル系ダイレクトライト API

関数名	ビットデータ
INT WINAPI WriteDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI WriteDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI WriteDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI WriteDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI WriteDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI WriteDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI WriteDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
関数名	文字列データ
INT WINAPI WriteDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI WriteDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI WriteDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI WriteSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI WriteSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ マルチハンドル系書き込み後キャッシュリフレッシュ付きライト API

関数名	ビットデータ
INT WINAPI WriteDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	16 ビットデータ
INT WINAPI WriteDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビットデータ
INT WINAPI WriteDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	16 ビット BCD データ
INT WINAPI WriteDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
関数名	32 ビット BCD データ
INT WINAPI WriteDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
関数名	単精度浮動小数点データ
INT WINAPI WriteDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
関数名	倍精度浮動小数点データ
INT WINAPI WriteDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
関数名	文字列データ
INT WINAPI WriteDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
関数名	汎用データ
INT WINAPI WriteDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
関数名	汎用データ（Variant 型）
INT WINAPI WriteDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
関数名	グループシンボル
INT WINAPI WriteSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
関数名	グループシンボル（Variant 型）
INT WINAPI WriteSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

※各パラメータについては、「■リード／ライト関数のパラメータ」を参照してください。

■ リード／ライト関数のパラメータ

＜引数＞

bsNodeName　　：接続機器名付き参加局名（文字列）へのポインタ

『Pro-Studio EX』で登録された参加局の局名、または IP アドレスを直接記述します。
例 1）局名で指定する場合 “ AGP ”
例 2）IP アドレスを直接指定する場合 “ 192.9.201.1 ”

bsDeviceName　　：Read ／ Write するシンボル（文字列）へのポインタ

『Pro-Studio EX』で登録されたシンボル名、またはデバイスアドレスを直接記述します。
例 1）シンボルで指定する場合　　“ SWITCH1 ”
例 2）デバイスアドレスを直接指定する場合“ M100 ”

Function	シンボルのデータタイプ							
	Bit	16Bit		32bit		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　　：Read ／ Write データへのポインタ

アクセスするデータの種類と、対応する引数のタイプは下表の通りです。

アクセスするデータの種類	引数のタイプ
ビットデータ	WORD ＊ pwData
16 **ビットデータ**	WORD ＊ pwData
32 **ビットデータ**	DWORD ＊ pdwData
16 **ビット** BCD **データ**	WORD ＊ pwData
32 **ビット** BCD **データ**	DWORD ＊ pdwData
単精度浮動小数点データ	FLOAT ＊ pflData
倍精度浮動小数点データ	DOUBLE ＊ pdbData
文字列データ	LPTSTR psData
汎用データ	LPVOID pData
汎用データ（VB **用）**	LPVARIANT pData

wCount　　：Read ／ Write データ数

Read/WriteDeviceStr 関数の場合、文字列データのデータ数は 1 バイト単位です。シンボルが 16 ビット幅のデバイスの場合は 2 文字、32 ビット幅のデバイスの場合は 4 文字単位で指定してください。
読み書きできる最大リード数／ライト数は下表のとおりです。

アクセスするデータの種類	リード時	ライト時
ビットデータ	255	255
16 **ビットデータ**	1020	1020
32 **ビットデータ**	510	510
16 **ビット** BCD **データ**	1020	1020
32 **ビット** BCD **データ**	510	510
単精度浮動小数点データ	510	510
倍精度浮動小数点データ	255	255
文字列データ	2040 文字（半角）	2040 文字（半角）

wAppKind　　：データタイプ指定

値	データタイプ	値	データタイプ
1	Bit	7	Unsigned 32 Bit
2	Signed 16 Bit	8	HEX 32 Bit
3	Unsigned 16 Bit	9	BCD 32 Bit
4	HEX 16 Bit	10	Float
5	BCD 16 Bit	11	Double
6	Signed 32 Bit	12	String

Read/WriteDevice 関数は、データタイプをパラメータで指定するので、動的にデータタイプを変更できます。

＜戻り値＞

正常終了：0

異常終了：エラーコード

＜補足＞

Read/WriteDeviceBit 関数を使用する場合

pwData には、wCount 数分だけ D0 ビットから詰めて格納します。

例：wCount が 20 の場合

	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

連続する複数ビットデータを扱う場合は、Read/WriteDeviceBit より Read/WriteDevice16 や Read/WriteDevice32 で 16/32 ビット単位の Read/Write の方が効率的です。

「*」には不定な値が入ります。アプリケーションプログラムでマスクしてください。

Read/WriteDeviceBCD16/32 関数を使用する場合

接続機器内部で、データを BCD として扱っている場合は、これらの関数を使用します。ただし、この関数と受け渡しするデータ（pxxData の内容）は、BCD ではなくバイナリデータとなります。

（『Pro-Server EX』内部で BCD 変換を行っています。）負の数は扱えません。

関数名	10 進表現	16 進表現
Read/WriteDeviceBCD16	0 ～ 9999	0000 ～ 270F
Read/WriteDeviceBCD32	0 ～ 99999999	00000000 ～ 05F5E0FF

文字列データ関数を使用する場合

文字列データを受け取る変数は、受け取れるだけの十分なデータ領域を確保してください。

27.3 キャッシュバッファ制御 API

関数名	キャッシュバッファの構築

『Pro-Server EX』はデバイスのリードを高速化するために、内部にデバイスのデータをキャッシュする機能（コピーを持つ機能）があります。この API はキャッシュを行うためのバッファを構築します。
この API はキャッシュバッファの器を定義するだけで、実際にどのデバイスをキャッシュするかの定義は、PS_EntryCacheRecord() で行います。

シングル
INT WINAPI PS_CreateCache(LPCSTR sCacheName, DWORD dwPollingTime);
マルチ
INT WINAPI PS_CreateCacheM(HANDLE hProServer, LPCSTR sCacheName, DWORD dwPollingTime);

引数
sCacheName：（In）キャッシュバッファ名
dwPollingTime：（In）0 を指定すると常時監視方式になり、できる限り高速にキャッシュを更新します。
0 以外なら ポーリング方式になり、そのポーリングの周期（キャッシュの更新周期）を ms 単位で指定してください。

戻り値
正常終了：0
異常終了：エラーコード

特記事項
- 1 台の『Pro-Server EX』に構築できるキャッシュバッファの数は最大 1000 個です。
- 『Pro-Studio EX』でネットワークプロジェクトファイル作成時に登録されたキャッシュバッファは、この API で新たに構築する必要はありません。直接利用できます。

関数名	キャッシュバッファへのレコード登録

PS_CreateCache() で作られたキャッシュバッファに、実際にキャッシュするデバイス（キャッシュ元デバイス）を登録します。

GP シリーズ局または Pro-Server EX 局はキャッシュの更新方法で、常時監視方式をサポートしていません。そのため、常時監視方式のキャッシュバッファ（PS_CreateCache() でキャッシュバッファを作成するとき、dwPollingTime に 0 を指定し作成したキャッシュバッファ）に対し、この API で GP シリーズ局または Pro-Server EX 局を指定するとエラーになります。

シングル
INT WINAPI PS_EntryCacheRecord(LPCSTR sCacheName, LPCSTR sNodeName, LPCSTR sDevice, WORD wAppKind, WORD wCount);
マルチ
INT WINAPI PS_EntryCacheRecordM(HANDLE hProServer, LPCSTR sCacheName, LPCSTR sNodeName, LPCSTR sDevice, WORD wAppKind, WORD wCount);

引数

sCacheName：(In) キャッシュバッファ名
このキャッシュバッファ名を持つキャッシュバッファに、キャッシュ元デバイスを登録します。

sNodeName：(In) キャッシュ元のデバイスがある接続機器名付き参加局名

sDevice：(In) キャッシュ元デバイス
キャッシュ元デバイスには、デバイスアドレスを直接指定する方法と『Pro-Studio EX』で登録されたシンボル、またはグループを指定する方法があります。グループを指定すると、複数のシンボルを一括して登録できます。

wAppKind：(In) 元デバイスのデータ型
キャッシュ元デバイスにどの方法でデバイスを指定したかにより、指定できる値が違います。

a) キャッシュ元デバイスにデバイスアドレスを直接指定した場合
『Pro-Server EX』で利用できるデータタイプ（1 から 12）を指定してください。0 は指定できません。

値	データタイプ	値	データタイプ
1	Bit	7	10 進符号無し 32 ビット
2	10 進符号付き 16 ビット	8	16 進 32 ビット
3	10 進符号無し 16 ビット	9	BCD 32 ビット
4	16 進 16 ビット	10	単精度浮動小数点
5	BCD 16 ビット	11	倍精度浮動小数点
6	10 進符号付き 32 ビット	12	文字列

b) キャッシュ元デバイスにシンボルを指定した場合
『Pro-Server EX』で利用できるデータタイプ（0 から 12）を指定してください。0 を指定すると、シンボル定義時に指定したシンボルの型が使用されます。

c) キャッシュ元デバイスにグループを指定した場合
0 固定
グループ内の全シンボルがそのシンボルの型で登録されます。

wCount：(In) キャッシュするデバイスのデータ数
キャッシュ元デバイスにどの方法でデバイスを指定したかにより、指定できる値が違います。

a) キャッシュ元デバイスにデバイスアドレスを直接指定した場合
デバイスの型に応じたデータ数（1 ～ 1020、型により最大値は違います。）

b) キャッシュ元デバイスにシンボルを指定した場合
0 を指定するとシンボル定義時に指定した個数が利用されます。
0 以外はデバイスの型に応じたデータ数（1 ～ 1020、型により最大値は違います。）

c) キャッシュ元デバイスにグループを指定した場合
0 固定
グループ内の全シンボルがキャッシュ対象となります。

戻り値

正常終了：0
異常終了：エラーコード

特記事項

関数名	キャッシュ動作の開始

キャッシュ動作の開始を行います。

シングル
INT WINAPI PS_StartCache(LPCSTR sCacheName);
マルチ
INT WINAPI PS_StartCacheM(HANDLE hProServer, LPCSTR sCacheName);

引数
sCacheName：（In）開始するキャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	キャッシュ動作の停止

キャッシュ動作を一時的に停止します。
キャッシュ動作が停止するだけで、キャッシュバッファの定義は残っています。
PS_StartCache() をコールすれば再開できます。

シングル
INT WINAPI PS_StopCache(LPCSTR sCacheName);
マルチ
INT WINAPI PS_StopCacheM(HANDLE hProServer, LPCSTR sCacheName);

引数
sCacheName：（In）停止するキャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	キャッシュ動作状況確認

キャッシュ動作状況の確認を行います。

シングル
INT WINAPI PS_GetCacheStatus(LPCSTR sCacheName);
マルチ
INT WINAPI PS_GetCacheStatusM(HANDLE hProServer, LPCSTR sCacheName);

引数
sCacheName：（In）確認するキャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。

戻り値
0：キャッシュバッファは構築されただけで、一度も開始されていません
1：キャッシュ動作稼動中
2：キャッシュ動作停止中
XX：エラーコード

特記事項

関数名	キャッシュバッファの破棄

キャッシュ動作を停止し、キャッシュバッファを破棄します。

シングル
INT WINAPI PS_DestroyCache(LPCSTR sCacheName);
マルチ
INT WINAPI PS_DestroyCacheM(HANDLE hProServer, LPCSTR sCacheName);

引数
sCacheName：（In）破棄するキャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。

戻り値
正常終了：0
異常終了：エラーコード

関数名	キャッシュ更新通知機能セット

キャッシュの更新を、指定されたウィンドウに知らせる機能をセットします。

アプリケーションからデバイスのキャッシュリードを行う場合、キャッシュデータが更新されていなければ、頻繁にデバイスのキャッシュリードを行っても変化はありません。
そこで、『Pro-Server EX』は、キャッシュデータが更新されたとき（常時監視方式の場合は監視対象のデバイスのうち一つでも変化したとき、ポーリング方式の場合は 1 ポーリングが完了したとき）、指定されたウィンドウにメッセージを送る機能があります。
このメッセージを受けたあとにデバイスのキャッシュリードを行うようにシステムを構築すると、効率のよいシステムになります。
この API は、どのキャッシュが更新されたときに、“ どのウィンドウ ” と “ どんなメッセージを送るのか " を『Pro-Server EX』にセットします。
セットが正常完了すると、今回セットした通知機能を識別するための ID を返します。

シングル
INT WINAPI PS_SetNotifyFromCache(LPCSTR sCacheName, HWND hWnd, UINT message, WPARAM WParam, LPARAM LParam, HANDLE* ohCacheNotifyID);
マルチ
INT WINAPI PS_SetNotifyFromCacheM(HANDLE hProServer, LPCSTR sCacheName, HWND hWnd, UINT message, WPARAM WParam, LPARAM LParam, HANDLE* ohCacheNotifyID);

引数
sCacheName：（In）キャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。
hWnd：（In）メッセージの送り先ウィンドウのウィンドウハンドル
message：（In）ウィンドウに送るメッセージ ID
wParam：（In）メッセージ ID と同時にウィンドウに送られる WPARAM の値
LParam：（In）メッセージ ID と同時にウィンドウに送られる LPARAM の値
ohCacheNotifyID：（Out）今回セットした通知機能を識別するための ID が返されます。

戻り値
正常終了：0
異常終了：エラーコード

特記事項
返されたハンドルは必要が無くなれば PS_KillNotifyFromCache() で破棄してください。
『Pro-Server EX』はキャッシュが更新されると、第 2 引数に指定された message を、第 3 引数に wParam を、第 4 引数に LParam を指定して hWnd に PostMessage() します。
PostMessage() の詳細については、Windows の API マニュアルを参照してください。

関数名	次回のキャッシュ更新通知受け付け

次のキャッシュの更新通知を受け付けます。

『Pro-Server EX』はキャッシュが更新されると指定されたウィンドウに通知する機能がありますが、この通知は一回行われると、この API が呼ばれるまで次にキャッシュが更新されても通知しません。通知先ルーチンで処理に時間がかかった場合、Pro-Server EX が次のキャッシュ更新のメッセージを送ってしまうと、通知先ルーチンで多重呼び出しが発生する可能性があるためです。（通知先ルーチンで通知処理が終わっていないのに、次の通知を受けると、通知先ルーチンは多重呼び出しになります。）
これを防ぐため、この API は、明示的に次のメッセージを送ってもよいことを、『Pro-Server EX』に知らせます。
通知先ルーチンの通知処理の最後にこの API を呼び出せば、連続してキャッシュの更新に応じて処理するシステムを構築できます。

シングル
INT WINAPI PS_AcceptNextNotifyFromCache(HANDLE hCacheNotifyID);
マルチ
INT WINAPI PS_AcceptNextNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

引数	**戻り値**
hCacheNotifyID：(In) 次回の通知を許可する通知機能の ID PS_SetNotifyFromCache() で取得した ID	正常終了：0 異常終了：エラーコード

特記事項

関数名	キャッシュ更新通知のキャンセル

キャッシュ更新時をウィンドウに知らせる機能をキャンセルします。

『Pro-Server EX』は以後、hCacheNotifyID に関係付けられたキャッシュバッファを更新しても、通知（ウィンドウにメッセージを送る行為）を行いません。

シングル
INT WINAPI PS_KillNotifyFromCache(HANDLE hCacheNotifyID);
マルチ
INT WINAPI PS_KillNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

引数	**戻り値**
hCacheNotifyID：(In) キャンセル対象の通知機能の ID PS_SetNotifyFromCache() で取得した ID	正常終了：0 異常終了：エラーコード

特記事項
この API は、『Pro-Server EX』が送ったメッセージがウィンドウに残っていても、それを取り出して破棄することは行いません。そのため、この API がコールする前に、『Pro-Server EX』が先にメッセージをウィンドウに送っていて、そのメッセージをアプリケーション側でウィンドウから取り出していない場合、この API コール後でも、ウィンドウからメッセージは取り出せます。（API コール後でも、タイミング的に通知先ルーチンが呼び出されることはあります。）

関数名	キャッシュ更新回数の取得

キャッシュバッファの更新回数を返します。

更新回数の変化をプログラムで監視することで、キャッシュが更新されたかどうか知ることができます。それを利用すれば、無駄なデバイスのキャッシュリードを省くことができます。（変化がないのに、デバイスのキャッシュリードをコールしても値に変化はありません。）

シングル
INT WINAPI PS_GetUpdateCounter(LPCSTR sCacheName, DWORD* odwCount);
マルチ
INT WINAPI PS_GetUpdateCounterM(HANDLE hProServer, LPCSTR sCacheName, DWORD* odwCount);

引数
sCacheName：（In）監視するキャッシュバッファ名
『Pro-Studio EX』で登録したキャッシュバッファ名も指定できます。
odwCount：（Out）キャッシュの更新回数
0 からスタートし、0 から 4294967295 までを無限にカウントします（4294967295 の次は 0 に戻ります）。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

27.4 キューイングアクセス制御 API

関数名	デバイスリード要求のキューイング開始

この API コール後、ExecuteQueuingAccess() がコールされるまでデバイスリード系の要求をキューイングします。
キューイングは、Pro-Server ハンドル単位で行われます。

シングル
INT WINAPI BeginQueuingRead();
マルチ
INT WINAPI BeginQueuingReadM(HANDLE hProServer);

引数	戻り値 正常終了：0 異常終了：エラーコード

特記事項
- BeginQueuingRead() がコールされてから ExecuteQueuingAccess() がコールされるまで、デバイスライト系の API をコールしないでください。以後のキャッシュリード、ダイレクトリードの命令はキューイングされます。ただし、キャッシュリードとダイレクトリードの命令を混在させることはできません。
- キューイング中の命令を破棄する場合は、CancelQueuingAccess() をコールしてください。
- キューイングできる最大命令数は 1500 件、データの最大バイト数は 1MByte 以内です。

関数名	デバイスライト要求のキューイング開始

この API コール後、ExecuteQueuingAccess() がコールされるまで、デバイスライト系の要求をキューイングします。
キューイングは Pro-Server ハンドル単位で行われます。

シングル
INT WINAPI BeginQueuingWrite();
マルチ
INT WINAPI BeginQueuingWriteM(HANDLE hProServer);

引数	戻り値 正常終了：0 異常終了：エラーコード

特記事項
- BeginQueuingWrite() がコールされてから ExecuteQueuingAccess() がコールされるまで、デバイスリード系の API をコールしないでください。以後のキャッシュライト、ダイレクトライトの命令はキューイングされます。ただし、キャッシュライトとダイレクトライトの命令を混在させることはできません。
- キューイング中の命令を破棄する場合は、CancelQueuingAccess() をコールしてください。
- キューイングできる最大命令数は 1500 件、データの最大バイト数は 1MByte 以内です。

特記事項

関数名	キューイングしているデバイスリード／ライト要求の実行

キューイングしているデバイスリード／ライト要求に従い、実際にデバイスデータへアクセスします。

シングル
INT WINAPI ExecuteQueuingAccess();
マルチ
INT WINAPI ExecuteQueuingAccessM(HANDLE hProServer);

引数	戻り値 正常終了：0 異常終了：エラーコード

特記事項
- ExecuteQueuingAccess() は、すべてのデバイスへのアクセスが成功すると成功を返し、一つでも失敗するとアクセスエラーを返します。デバイスごとにアクセスの成否が知りたい場合は、IsQueuingAcceessSucceeded() をコールし確認します。
- キューイングアクセスにアクションを登録することはできません。

関数名	キューイングしているデバイスリード／ライト要求の破棄

キューイングしているデバイスリード／ライト要求を破棄します。

シングル
INT WINAPI CancelQueuingAccess();
マルチ
INT WINAPI CancelQueuingAccessM(HANDLE hProServer);

引数	戻り値 正常終了：0 異常終了：エラーコード

特記事項
BeginQueuingWrite() または BeginQueuingRead() をコール後、ExecuteQueuingAccess() をコールするまでは、デバイスへのアクセスの要求はキューイングされます。
もし、何らかの理由で、キューイングされている要求が不要になった場合は、この API をコールしてください。キューイングしている要求を破棄し、キューイング動作を終了します。

関数名	キューイングしているデバイスリード／ライト要求の破棄

ExecuteQueuingAccess() をコールしたあと、ここのデバイスアクセスへの要求の成否を求めます。

シングル
INT WINAPI IsQueuingAccessSucceeded(INT iIndex);
マルチ
INT WINAPI IsQueuingAccessSucceededM(HANDLE hProServer,INT iIndex);

引数
iIndex：（In）確認したい要求の番号

BeginQueuingWrite() または BeginQueuingRead() をコールして、ExecuteQueuingAccess() をコールするまでに、デバイスアクセスの要求をキューイングするために、デバイスアクセス系の API を何回かコールします。ただし、実際のデバイスアクセスの結果は、ExecuteQueuingAccess() 実行後でないと分かりません。
デバイスアクセスの結果を知りたい場合、ExecuteQueuingAccess() 実行後、その知りたいデバイスを要求した順番（0 からの番号）で指定してください。

戻り値
XX ：エラーコード
0 ：指定された番号のデバイスアクセスは成功しています。

特記事項
（例）

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

上記の登録で、"Node1" の "LS200" へのアクセスが成功しかたどうかは、IsQueuingAccessSucceeded(1) で確認します。
0 が返ればアクセスは成功しています。

27.5 システム系 API

関数名	Pro-Server ハンドルの生成
マルチハンドル系の関数を利用するときに使用する、Pro-Server のハンドルを取得します。 HANDLE WINAPI CreateProServerHandle();	
引数	**戻り値** 正常終了：0 以外（ハンドルコード） 異常終了：0
特記事項	

関数名	Pro-Server ハンドルの開放
取得済みの Pro-Server のハンドルを解放します。 INT WINAPI DeleteProServerHandle(HANDLE hProServer);	
引数 hProServer：（In）解放する Pro-Server のハンドル	**戻り値** 正常終了：0 異常終了：エラーコード
特記事項	

関数名	ネットワークプロジェクトファイルのロード
引数で指定されたネットワークプロジェクトをロードします。 シングル INT WINAPI EasyLoadNetworkProject(LPCSTR sDBName,DWORD dwSetOrAdd = TRUE); マルチ INT WINAPI EasyLoadNetworkProjectM(HANDLE hProServer,LPCSTR sDBName,DWORD dwSetOrAdd = TRUE);	
引数 sDBName：ロードするネットワークプロジェクトファイルをフルパスで指定します。 dwSetOrAdd：予約（1 固定） hProServer ：Pro-Server のハンドル	**戻り値** 正常終了：0 異常終了：エラーコード
特記事項	

関数名	エラーコードの文字列変換

『Pro-Server EX』の各種 API が返したエラーコードをエラーメッセージに変換します。
EasyLoadErrorMessage() は、メッセージとしてマルチバイト文字列（ASCII）を返します。
EasyLoadErrorMessageW() は、メッセージとしてワイド文字列（UNICODE）を返します。

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

引数
iErrorCode：（In）『Pro-Server EX』の関数が返したエラーコード
osErrorMessage：（Out）変換された文字列（マルチバイト文字列）を格納する領域へのポインタ（512 バイト以上を確保しコールしてください。）
owsErrorMessage：（Out）変換された文字列（ワイド文字列）を格納する領域へのポインタ（1024 バイト以上を確保しコールしてください。）

戻り値
正常終了：0 以外
文字列変換に失敗（使用されていないエラーコードなど）：0

特記事項
- この API は旧 Pro-Server との互換性のためにあります。
- EasyLoadErrorMessageEx() の方が、より詳しいエラーメッセージに変換しますので、そちらをご利用ください。

関数名	エラーコードの文字列変換（状況情報付き）

『Pro-Server EX』の各種 API が返したエラーコードをエラーメッセージに変換します。
このとき、可能であれば、エラーが発生した状況や情報を付加してエラーメッセージを返します。
EasyLoadErrorMessage() は、指定されたエラーコードに対して常に同じエラーメッセージを返しますが、EasyLoadErrorMessageEx() は、通信相手名やエラーの発生場所など、エラーの発生した状況により、より詳しいエラー情報を返します。同じエラーコードでも場合によっては違うエラーメッセージを返します。
EasyLoadErrorMessageEx(),EasyLoadErrorMessageExM() は、メッセージとしてマルチバイト文字列（ASCII）を返します。
EasyLoadErrorMessageExW(),EasyLoadErrorMessageExWM() は、メッセージとしてワイド文字列（UNICODE）を返します。

シングル
BOOL WINAPI EasyLoadErrorMessageEx(INT iErrorCode,LPSTR osErrorMessage);
BOOL WINAPI EasyLoadErrorMessageExW(INT iErrorCode,LPWSTR owsErrorMessage);
マルチ
BOOL WINAPI EasyLoadErrorMessageExM(HANDLE hProServer,INT iErrorCode,LPSTR osErrorMessage);
BOOL WINAPI EasyLoadErrorMessageExWM(HANDLE hProServer,INT iErrorCode,LPWSTR owsErrorMessage);

引数
iErrorCode :（In）『Pro-Server EX』の関数が返したエラーコード
osErrorMessage :（Out）変換された文字列（マルチバイト文字列）を格納する領域へのポインタ（1024 バイト以上を確保しコールしてください。）
owsErrorMessage :（Out）変換された文字列（ワイド文字列）を格納する領域へのポインタ（2048 バイト以上を確保しコールしてください。）

戻り値
正常終了：0 以外
文字列変換に失敗（使用されていないエラーコードなど）：0

特記事項
- EasyLoadErrorMessage() は、『Pro-Server EX』の API をコールして、その API がエラーコードを返した場合、そのエラーコードをメッセージに変換するために利用される場合を想定しています。
- 『Pro-Server EX』は、ハンドルごとにエラーの状況情報を 1 つ分しか覚えていません。そのため、エラーの発生元となった API と EasyLoadErrorMessage() の間に、別の API をコールするとエラーの状況情報が書き換えられてしまうため、EasyLoadErrorMessage() はエラーの状況情報を返しません。同じ理由で、EasyLoadErrorMessageM() を利用するときは、エラーの発生元となった API をコールしたときに使用した Pro-Server ハンドルで、EasyLoadErrorMessageM() をご使用ください。

関数名	Pro-Server API の初期化

Pro-Server EX API を初期化し、利用を内部的に宣言します。
『Pro-Server EX』を起動せずに EasyInit() を実行すると、『Pro-Server EX』を自動的に起動します。

INT WINAPI EasyInit();

引数	**戻り値** 正常終了：0 異常終了：エラーコード

特記事項

関数名	Pro-Server API の利用終了

INT WINAPI EasyTerm();

引数	**戻り値**

特記事項
この API は旧 Pro-Server との互換性のためにあります。
『Pro-Server EX』では、この API をコールする必要はありません。（コールしても実行しません。）

関数名	Pro-Server EX の終了

『Pro-Server EX』を終了させます。
この API をコールしたあと、『Pro-Server EX』の API はコールしないでください。
Pro-Server ハンドルなど、この API のコール前に必ず破棄してください。

INT WINAPI EasyTermServer();

引数	**戻り値** 正常終了：0 異常終了：エラーコード

特記事項

関数名	Pro-Server EX の終了通知

この API を利用すると、『Pro-Server EX』の終了を知ることができます。
『Pro-Server EX』は終了処理を始めると、この API で登録されているウィンドウに指定されたメッセージを、Windows の API の PostMessage() を利用して送ります。
PostMessage() の詳細については、Windows の API を参照してください。
アプリケーションは、ウィンドウからメッセージを受け取ることで、『Pro-Server EX』が終了の直前であることを認識できます。

シングル
INT WINAPI EasyNotifyFromServerEnd(HWND hReceivedWnd,UINT uMessage,WPARAM WParam = 0 , LPARAM LParam = 0);
マルチ
INT WINAPI EasyNotifyFromServerEndM(HANDLE hProServer,HWND hReceivedWnd,UINT uMessage,WPARAM WParam = 0 , LPARAM LParam = 0);

引数
hReceivedWnd：（In）終了メッセージを受け取るウィンドウ
uMessage：（In）終了メッセージとして送るメッセージ ID
この ID が『Pro-Server EX』終了時に、hReceivedWnd に送られます
WParam：（In）メッセージと同時に渡される WPARAM（PostMessage() の WPARAM 値）
LParam：（In）メッセージと同時に渡される LPARAM（PostMessage() の LPARAM 値）

戻り値
正常終了：0
異常終了：エラーコード

特記事項
『Pro-Server EX』の終了と同時に終了するアプリケーションを構築する場合、この API を利用すると便利です。
例えば、hReceivedWnd にアプリケーションのメインウィンドウを指定し、uMessage に WM_QUIT を指定してこの API をコールすると、『Pro-Server EX』終了時に、アプリケーションのメインウィンドウに WM_QUIT が送られます。
通常、アプリケーションは WM_QUIT をアプリケーションの終了の合図として利用しているため、『Pro-Server EX』が終了すると、終了するアプリケーションが構築できます。

関数名	メッセージ処理の抑制

多くの Pro-Server EX API の関数は処理に時間がかかる場合、関数の中で Windows のメッセージを処理していますが、この Windows のメッセージ処理を行うか、抑制するかを指定することができます。
抑制にすると、関数実行中は Windows のメッセージをメッセージキューに蓄積したまま処理しません。
その結果、関数処理中にアイコンがクリックされ、関数の二重呼び出しを行ってしまうようなことが起こらなくなります。
ただし、この場合、「アイコンがクリックされた」というメッセージだけではなく、全ての Windows のメッセージ処理が抑制され、タイマーやウィンドウの再描画など、重要なメッセージの処理が行われませんのでご注意ください。
処理を行うか抑制するかは、『Pro-Server EX』のハンドルごとに指定できます、デフォルトは「処理する」になっています。

シングル
INT EasySetWaitType(DWORD dwMode);
マルチ
INT EasySetWaitTypeM(HANDLE hProServer,DWORD dwMode);

引数
hProServerHandle：（In）処理モードを変更する Pro-Server のハンドル
dwMode：（In）1 を指定するとメッセージの処理を行います。
2 を指定するとメッセージの処理を抑制します。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	メッセージ処理方法の取得

Pro-Server EX API コール中のメッセージ処理方法が、現在どのモードになっているかを取得する関数です。
マルチハンドルの場合は、ハンドルごとに現在のモードを返します。

シングル
INT EasyGetWaitType();
マルチ
INT EasyGetWaitTypeM(HANDLE hProServerHandle);

引数
HANDLE hProServerHandle：（In）状態を取得するハンドル

戻り値
1：メッセージの処理を行います。
2：メッセージの処理を抑制します。

特記事項

関数名	ログビューアにログを追加する

『Pro-Server EX』は、内部動作で特定の事象（『Pro-Server EX』の起動や終了、エラーなど）が発生した場合、それを記録する機能があります。
記録された情報は、ログビューアを利用して見ることができます。（「28.5　システム稼動ログが見たい！」参照）
この API はその機能を利用し、特定のメッセージを記録します。アプリケーションのデバックなどにご利用ください。

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

引数

bLevel：（In）事象の種類

すべてのメッセージを記録しているとパフォーマンスが低下する場合があります。そのため、記録するメッセージを事象の種類ごとにフィルタリングする機能があります。
今回記録するメッセージがどの事象に属するかを指定します。
以下に事象の種類を示します。

定義	16 進値	事象の種類
EASY_LogLevel_SysMessage	0x01	システムメッセージ
EASY_LogLevel_SysError	0x02	システムエラーメッセージ
EASY_LogLevel_AppError	0x04	ユーザープログラムのエラーメッセージ
EASY_LogLevel_AppStart	0x08	ユーザープログラムの開始メッセージ
EASY_LogLevel_AppEnd	0x10	ユーザープログラムの終了メッセージ
EASY_LogLevel_AppWarning	0x20	ユーザープログラムの警告メッセージ
EASY_LogLevel_AppMessage1	0x40	ユーザープログラムの詳細メッセージ 1
EASY_LogLevel_AppMessage2	0x80	ユーザープログラムの詳細メッセージ 2

sPrompt：（In）事象の発生箇所を示す文字列（NULL 終端）
sMessage：（In）記録するメッセージ文字列（NULL 終端）

実際に記録されるメッセージは sPrompt と sMessage の二つを単純に連結した文字列が記録されます。

戻り値

正常終了：0
異常終了：エラーコード

特記事項

関数名	ログビューアのログをクリアする
EasyOutputLog() で記録した情報をクリアします。 この API はアプリケーションのデバックにご利用ください。 INT WINAPI EasyOutputLogClear();	
引数 HANDLE hProServerHandle：（In）状態を取得するハンドル	**戻り値** 正常終了：0 異常終了：エラーコード
特記事項	

27.6 SRAM 内データアクセス API

関数名	SRAM バックアップデータの読み出し

GP シリーズ局の SRAM 内にある下記データを読み出し、パソコン内のファイルとして保存します。
保存されるファイル形式は　ファイリングデータの場合はバイナリ形式のファイル、それ以外のデータの場合は CSV 形式のファイルです。

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

引数

sSaveFileName：（In）読み出したデータの保存先ファイルのファイルパス（文字列のポインタ）

sNodeName：（In）読み出すデータ元の参加局名（文字列のポインタ）
Pro-Server EX 局は指定できません。

iSaveMode：（In）保存方法
0：新規（同名のファイルがすでにある場合は、そのファイルを一旦削除し、上書きします。）
1：追加（ファイルの最後に追記する、まだそのファイルが無い場合は、新規に作成します。）
上記以外：予約

iBackupDataType：（In）読み出すデータの種類

値	データ元局が GP シリーズ局	データ元局が GP3000 シリーズ局、WinGP 局および LT3000 局
0x0001	ファイリングデータ	ファイリングデータ
0x0002	ロギングデータ	サンプリンググループ番号 1 のサンプリングデータ
0x0003	折れ線グラフデータ	サンプリンググループ番号 1 以外の全てのサンプリンググループのデータ
0x0004	サンプリングデータ	
0x0005	アラームブロック 1	アラームブロック 1
0x0006	アラームヒストリまたはアラームブロック 2	アラームブロック 2
0x0007	アラームログまたはアラームブロック 3	アラームブロック 3
0x0008	アラームブロック 4	アラームブロック 4
0x0009	アラームブロック 5	アラームブロック 5
0x000A	アラームブロック 6	アラームブロック 6
0x000B	アラームブロック 7	アラームブロック 7
0x000C	アラームブロック 8	アラームブロック 8
上記以外	（予約）	（予約）

データ元局が GP3000 シリーズ局、WinGP 局および LT3000 局で、データの種類がアラームブロック 1 ～ 8 の場合、1 アラームブロック内には『GP-Pro EX』の設定により最大アクティブデータ、ヒストリデータ、ログデータの 3 種類が格納されていますが、この API では、以下の優先順位で有効なデータを持っているか確認し、あればそれを対象とします。
①アラームヒストリ
②アラームログ
③アラームアクティブ
どれも有効でなければエラーとなります。

戻り値

正常終了：0
異常終了：エラーコード

特記事項

関数名	SRAM バックアップデータの拡張読み出し

GP シリーズ局の SRAM 内にある下記データを読み出し、パソコン内のファイルとして保存します。
保存されるファイル形式は　ファイリングデータの場合はバイナリ形式のファイル、それ以外のデータの場合は CSV 形式のファイルです。
EasyBackupDataRead() にくらべ、GP3000 シリーズ局、WinGP 局および LT3000 局用に拡張されたデータにアクセスすることができます。

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

引数

sSaveFileName：（In）読み出したデータの保存先ファイルのファイルパス（文字列のポインタ）

sNodeName：（In）読み出すデータ元の参加局名（文字列のポインタ）
Pro-Server EX 局は指定できません。

iSaveMode：（In）保存方法
0：新規（同名のファイルがすでにある場合は、そのファイルを一旦削除し、上書きします。）
1：追加（ファイルの最後に追記する、まだそのファイルが無い場合は、新規に作成します。）
上記以外：予約

iBackupDataType：（In）読み出すデータの種類

値	データ元局が GP シリーズ局	データ元局が GP3000 シリーズ局 ,WinGP 局 および LT3000 局
0x0001	ファイリングデータ	ファイリングデータ
0x0002	ロギングデータ	サンプリンググループ番号 1 のサンプリングデータ
0x0003	折れ線グラフデータ	サンプリンググループ番号 1 以外の全てのサンプリンググループのデータ
0x0004	サンプリングデータ	
0x0005	アラームブロック 1	アラームブロック 1 アラームの種類は iNumber で指定します
0x0006	アラームヒストリまたは アラームブロック 2	アラームブロック 2 アラームの種類は iNumber で指定します
0x0007	アラームログまたはア ラームブロック 3	アラームブロック 3 アラームの種類は iNumber で指定します
0x0008	アラームブロック 4	アラームブロック 4 アラームの種類は iNumber で指定します
0x0009	アラームブロック 5	アラームブロック 5 アラームの種類は iNumber で指定します
0x000A	アラームブロック 6	アラームブロック 6 アラームの種類は iNumber で指定します
0x000B	アラームブロック 7	アラームブロック 7 アラームの種類は iNumber で指定します
0x000C	アラームブロック 8	アラームブロック 8 アラームの種類は iNumber で指定します
0x8002	（予約）	特定のグループ番号のサンプリンググループ グループ番号は iNumber で指定します

戻り値

正常終了：0
異常終了：エラーコード

iNumber:（In）この引数は、sSaveFileName が GP3000 シリーズ局、WinGP 局および LT3000 局の場合有効で、GP シリーズ局の場合は無視されます。
さらに、iBackupDataType の値により意味が異なります。

iBackupDataType の値	内容
0x0005 から 0x000C	アラームデータの種類にはアクティブ、ヒストリ、ログの 3 種類がありますが、どれを対象にするかを指定します。 iNumber の値 / 内容 0：以下の優先順位でアラームブロックが有効なデータを持っているか確認し、あればそれを対象とします。 ①アラームヒストリ ②アラームログ ③アラームアクティブ どれも有効でなければエラーとなります。 1：アラームアクティブを対象とします。 2：アラームヒストリを対象とします。 3：アラームログを対象とします。 もし対象としたデータの種類を iBackupDataType で指定されたアラームブロックが持っていない場合はエラーとなります。
0x8002	読み出す対象のサンプリンググループのグループ番号 1 から 64 の値
上記以外	（予約）

iStringTable：（In）予約
常に 0 を指定してください。

関数名	SRAM バックアップデータの書き込み

指定されたバイナリ形式のファイリングデータを、GP シリーズ局の SRAM 内に書き込みます。

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

引数

sSourceFileName：（In）書き込むバイナリ形式のファイリングデータファイルのファイルパス（文字列のポインタ）

sNodeName：（In）書き込み先の参加局名（文字列のポインタ）
Pro-Server EX 局、GP3000 シリーズ局、WinGP 局、LT3000 局は指定できません。

BackupDataType：（In）「1」固定（ファイリングデータを意味します）

戻り値

正常終了：0
異常終了：エラーコード

特記事項

27.7 CF カード関係 API

MEMO
- LT3000 シリーズおよび GP3200 には CF カードスロットがないため、CF 関係 API を使用することができません。

関数名	CF カードステータス読み出し

接続されている機器（GP）の CF カード接続状態を取得できます。

シングル
INT WINAPI EasyIsCFCard(LPCSTR sNodeName) ;
マルチ
INT WINAPI EasyIsCFCardM(HANDLE hProServer,LPCSTR sNodeName) ;

引数
hProServer：Pro-Server ハンドル
sNodeName：読み出し先 GP の局名（この局名はすでにネットワークプロジェクトに登録されている必要があります。）

戻り値

関数の戻り値	GP シリーズ局の場合	GP3000 シリーズ局および WinGP 局の場合
0x00000000	正常	正常
0x10000001	CF カードなし	CF カードなし、または CF カードスロットのカバーが開いている（CF カード有無は関係なし）
0x10000002	CF カードドライバでサポートできないデバイスを検出	
0x10000004	CF カード異常を検出	CF カード異常を検出
0x10000008	CF カード未初期化	
その他	CF カード関連以外のエラー	

特記事項

関数名	CF カード内ファイル一覧読み出し（任意フォルダ名）

GP に挿入されている CF カード内にあるファイル一覧をパラメータで渡されたファイルに出力します。
ファイル一覧を取得したいフォルダを任意に指定できます。

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

引数

sNodeName：読み出し先 GP の局名
sDirectory：取得するフォルダ名（すべて大文字）
oiCount：読み出したファイルの数
sSaveFileName：読み出したディレクトリ情報の格納先ファイル名。なお、指定したファイル内には、stEasyDirInfo 型の配列に格納されたデータが、pioCount で返された個数分、バイナリデータとして格納されます。なお、ファイル名、ファイルの拡張子はすべて大文字として保存されます。

```
struct stEasyDirInfo {
   BYTE bFileName[8+1];// ファイル名 ( 最後は 0 で完結 )
  BYTE bExt[3+1];// ファイルの拡張子 ( 最後は 0 で完結 )
  BYTE bDummy[3];// ダミー
  DWORD dwFileSize;// ファイルのサイズ
  BYTE bFileTimeStamp[8+1];// ファイルのタイムスタンプ ( 最後は0 で完結 )
 BYTE bDummy2[3];// ダミー 2
 } ;
```

戻り値

正常終了：0
異常終了：エラーコード

特記事項

「bFileTimeStamp」の補足として、8 バイトのうち、上位 4 バイトが MS-DOS 形式の時刻を、下位 4 バイトが MS-DOS 形式の日付を 16 進文字列として表しています。
なお、MS-DOS 形式の日付、時刻のフォーマットは以下のとおりです。
(例：20C42C22 の場合、2C22 が MS-DOS の日付を 16 進で表記したもの、20C4 が MS-DOS の時刻を 16 進で表したものとなるため、2002/1/2 4:6:8 を表すことになります。)

ビット	内容
0 〜 4	年月日の日（1 〜 31）。
5 〜 8	年月日の月（1=1 月、2=2 月、〜 12=12 月）。
9 〜 15	年月日の年。ただし、1980 年からの経過年数で指定します。これらのビットが表す値に 1980 を足すと、実際の年が得られます。

MS-DOS 形式の時刻を指定します。この日付は、次の形式で 1 個の 16 ビット値にパックされています。

ビット	内容
0 〜 4	時分秒の秒を 2 で割った値です（0 〜 29）。
5 〜 10	時分秒の分（0 〜 59）。
11 〜 15	時分秒の時（24 時間制で 0 〜 23）。

GP3000 シリーズ局、WinGP 局、または GP シリーズ局から読み出したファイル一覧の中に、ファイル名が 8 文字に満たない場合または拡張子名が 3 文字に満たないファイルが含まれる場合、bFileName[8+1] および bExt[3+1] は、以下のように表示されます。

読み出し元局	GP3000 シリーズ局または WinGP 局	GP シリーズ局
bFileName[8+1] の表示	ファイル名が 8 文字に満たない部分の先頭に 0 が格納され、その後は不定値が格納される	ファイル名が 8 文字に満たない部分は半角スペース (0x20) が格納され、最後は 0 が格納される
bExt[3+1] の表示	拡張子名が 3 文字に満たない部分の先頭に 0 が格納され、その後は不定値が格納される	拡張子名が 3 文字に満たない部分は半角スペース (0x20) が格納され、最後は 0 が格納される

（例）ファイル名および拡張子名が ABC.D の場合
GP3000 シリーズ局または WinGP 局

bFileName[8+1] の表示	0x410x420x430x00********** (**** は不定値)
bExt[3+1] の表示	0x440x00******** (**** は不定値)

GP シリーズ局

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

関数名	CF カード内ファイル一覧読み出し（タイプ指定）

GP に挿入されている CF カード内にあるファイル一覧をパラメータで渡されたファイルに出力します。読み出すファイル一覧は “ sDirectory ” で指定するディレクトリに限ります。

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

引数

sNodeName：読み出し先 GP の局名

sDirector：取得するディレクトリ名 (全て大文字)。なお、以下のディレクトリのみサポートします。

- LOG（ロギングデータ）
- TREND（トレンドデータ）
- ALARM（アラームデータ）
- CAPTURE（キャプチャデータ）
- FILE（ファイリングデータ）

oiCount：読み出したファイルの数

sSaveFileName：読み出したディレクトリ情報の格納先ファイル名。なお、指定したファイル内には、stEasyDirInfo 型の配列に格納されたデータが、pioCount で返された個数分、バイナリデータとして格納されます。なお、ファイル名、ファイルの拡張子はすべて大文字として保存されます。

```
struct stEasyDirInfo {
    BYTE bFileName[8+1];// ファイル名 ( 最後は 0 で完結 )
    BYTE bExt[3+1];// ファイルの拡張子 ( 最後は 0 で完結 )
    BYTE bDummy[3];// ダミー
    DWORD dwFileSize;// ファイルのサイズ
    BYTE bFileTimeStamp[8+1];// ファイルのタイムスタンプ ( 最後は 0 で完結 )
    BYTE bDummy2[3];// ダミー 2
    } ;
```

戻り値

正常終了：0

異常終了：エラーコード

特記事項
GP3000 シリーズ局、WinGP 局、または GP シリーズ局から読み出したファイル一覧の中に、ファイル名が 8 文字に満たない場合または拡張子名が 3 文字に満たないファイルが含まれる場合、bFileName[8+1] および bExt[3+1] は、以下のように表示されます。

読み出し元局	GP3000 シリーズ局または WinGP 局	GP シリーズ局
bFileName[8+1] の表示	ファイル名が 8 文字に満たない部分の先頭に 0 が格納され、その後は不定値が格納される	ファイル名が 8 文字に満たない部分は半角スペース (0x20) が格納され、最後は 0 が格納される
bExt[3+1] の表示	拡張子名が 3 文字に満たない部分の先頭に 0 が格納され、その後は不定値が格納される	拡張子名が 3 文字に満たない部分は半角スペース (0x20) が格納され、最後は 0 が格納される

（例）ファイル名および拡張子名が ABC.D の場合
GP3000 シリーズ局または WinGP 局

bFileName[8+1] の表示	0x410x420x430x00********** (**** は不定値)
bExt[3+1] の表示	0x440x00******** (**** は不定値)

GP シリーズ局

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

関数名	CF カードファイル読み出し（任意ファイル名指定）

CF カードに存在する指定されたファイルの内容を読み出す関数です。読み出すファイルを任意に指定できます。

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

引数
sNodeName：読み出し先 GP の局名
sFolderName：読み出す CF カード内のファイルのフォルダ名（最大文字数半角 32 文字）
sFileName：読み出す CF カード内のファイル名（最大 8.3 形式の文字列）
pWriteFileName：読み出した CF ファイルの保存先ファイル名（フルパス）
odwFileSize：読み出した CF ファイルのファイルサイズ

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	CF カード内ファイル読み出し（タイプ指定）

CF カードに存在する指定されたファイルの内容を読み出す関数です。読み出すファイルは“ pReadFileType ”で指定するファイルの種類に限ります。

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

引数
sNodeName：読み出し先 GP の局名
pReadFileType：読み出す CF カード内のファイルの種類（＜特記事項＞参照）
wReadFileNo：読み出す CF カード内のファイル番号
sWriteFileName：読み出した CF ファイルの保存先ファイル名（フルパス）
odwFileSize：読み出した CF ファイルのファイルサイズ

戻り値
正常終了：0
異常終了：エラーコード

特記事項
サポートしているファイルの種類は以下の表のとおりです。CF カードの指定フォルダ内に保存されているもののみ、読み込み可能です。

■ GP シリーズ局でサポートするファイルの種類

データ種別	ファイルの種類	対象フォルダ
ファイリングデータ	ZF	FILE
CSV **データ**	ZR	FILE
イメージ画面	ZI	DATA
サウンドデータ	ZO	DATA
折れ線データ	ZT	TREND
サンプリング	ZS	TREND
アラーム 4 ～ 8	Z4 ～ Z8	ARAM
ロギングデータ	ZL	LOG
アラームログ	ZG	ALARM
アラームヒストリ	ZH	ALARM
アラームアクティブ	ZA	ALARM
画面データバックアップ	ZC	MRM
画面キャプチャ	CP	CAPTURE

■ GP3000 シリーズ局および WinGP 局でサポートするファイルの種類

データ種別	ファイルの種類	対象フォルダ
ファイリングデータ	ZF または F	FILE
CSV **データ**	ZR	FILE
イメージ画面	ZI または I	DATA
サウンドデータ	ZO または O	DATA
『GP-Pro EX』**専用折れ線グラフデータ（互換用）**	ZT	TREND
『GP-Pro EX』**専用データサンプリングのデータ（互換用）**	ZS	TREND
アラーム 1	Z1 または ZA	ALARM
アラーム 2	Z2 または ZH	ALARM
アラーム 3	Z3 または ZG	ALARM
アラーム 4 ～ 8	Z4 ～ Z8	ALARM
『GP-Pro EX』**専用ロギングデータ（互換用）**	ZL	LOG
キャプチャデータ	CP	CAPTURE
サンプリング 1 ～ 64	ZS1 ～ ZS64	SAMP01 ～ SAMP64

関数名	CF カードファイル書き込み（任意ファイル名指定）

指定されたファイルを CF カードへ書き込む関数です。書き込むファイルを任意に指定できます。

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

引数

sNodeName：書き込み先 GP の局名
pReadFileName：CF カードへの書き込み元となるファイル名（フルパス）
sFolderName：CF カード内へ書き込むファイルのフォルダ名（最大文字数半角 32 文字）
sFileName：CF カード内へ書き込むファイル名（最大 8.3 形式の文字列）

戻り値

正常終了：0
異常終了：エラーコード

特記事項

関数名	CF カードファイル書き込み（タイプ指定）

指定されたファイルを CF カードへ書き込む関数です。書き込むファイルは “ pWriteFileType ” で指定するファイルの種類に限ります。

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

引数	**戻り値**
sNodeName：書き込み先 GP の局名 pReadFileName：CF カードへの書き込み元となるファイル名（フルパス） sWriteFileType：CF カード内へ書き込むファイルの種類 （CF カードファイル読み出し関数（タイプ指定）の＜特記事項＞を参照） wWriteFileNo：CF カード内へ書き込むファイルの番号	正常終了：0 異常終了：エラーコード

特記事項

関数名	CF カードファイル削除（任意ファイル指定）

CF カードに存在する指定されたファイルを削除する関数です。削除するファイルを任意に指定できます。

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

引数	**戻り値**
sNodeName：削除するファイルが存在する GP の局名 sFolderName：削除する CF カード内のファイルのフォルダ名（最大文字数半角 32 文字) sFileName：削除する CF カード内のファイル名（最大 8.3 形式の文字列）	正常終了：0 異常終了：エラーコード

特記事項

関数名	CF カードファイル削除（タイプ指定）

CF カードに存在する指定されたファイルを削除する関数です。削除するファイルは“ pDeleteFileType ”で指定するファイルの種類に限ります。

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

引数
sNodeName：削除するファイルが存在する GP の局名
pDeleteFileType：削除する CF カード内のファイルの種類（＜特記事項＞参照）
wDeleteFileNo：削除する CF カード内のファイル番号

戻り値
正常終了：0
異常終了：エラーコード

特記事項
存在しないファイルに対してこの関数を実行した場合は、エラーにならず正常終了します。
サポートしているファイルの種類は以下の表のとおりです。CF カードの指定フォルダ内に保存されているもののみ、読み込み可能です。

■ GP シリーズ局でサポートするファイルの種類

データ種別	ファイルの種類	対象フォルダ
ファイリングデータ	ZF	FILE
CSV **データ**	ZR	FILE
イメージ画面	ZI	DATA
サウンドデータ	ZO	DATA
折れ線データ	ZT	TREND
サンプリング	ZS	TREND
アラーム 4 ～ 8	Z4 ～ Z8	ARAM
ロギングデータ	ZL	LOG
アラームログ	ZG	ALARM
アラームヒストリ	ZH	ALARM
アラームアクティブ	ZA	ALARM
画面データバックアップ	ZC	MRM
画面キャプチャ	CP	CAPTURE

■ GP3000 シリーズ局および WinGP 局でサポートするファイルの種類

データ種別	ファイルの種類	対象フォルダ
ファイリングデータ	ZF または F	FILE
CSV **データ**	ZR	FILE
イメージ画面	ZI または I	DATA
サウンドデータ	ZO または O	DATA
『GP-Pro EX』**専用折れ線グラフデータ（互換用）**	ZT	TREND
『GP-Pro EX』**専用データサンプリングのデータ（互換用）**	ZS	TREND
アラーム 1	Z1 または ZA	ALARM
アラーム 2	Z2 または ZH	ALARM
アラーム 3	Z3 または ZG	ALARM
アラーム 4 ～ 8	Z4 ～ Z8	ALARM
『GP-Pro EX』**専用ロギングデータ（互換用）**	ZL	LOG
キャプチャデータ	CP	CAPTURE
サンプリング 1 ～ 64	ZS1 ～ ZS64	SAMP01 ～ SAMP64

関数名	CF カードファイル名変更

CF カードに存在する指定されたファイルを名称変更する関数です。

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

引数
sNodeName：書き込み先 GP の局名
sFolderName：CF カード内の名前変更するファイルのフォルダ名（最大文字数 半角 32 文字）
sFileName：CF カード内の名前変更されるファイル名（最大 8.3 形式の文字列）
sFileRename：名前変更後のファイル名（最大 8.3 形式の文字列）

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	CF カード空き容量取得

指定された参加局に接続されている CF カード内の空き容量を取得する関数です。

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

引数	**戻り値**
sNodeName：読み出し先 GP の局名 oiUnallocated：CF カード内の空き容量（バイト単位として取得される）	正常終了：0 異常終了：エラーコード

特記事項

関数名	FTP パッシブモード設定

『Pro-Server EX』は、GP シリーズ局の CF カードへのアクセスは専用プロトコルで行いますが、GP3000 シリーズ局および WinGP 局へのアクセスは FTP プロトコルで通信を行います。
『Pro-Server EX』の FTP プロトコルは通常モードとパッシブポートの 2 モードをサポートしています。
この API はそのモードを設定します。

INT WINAPI EasyFileSetPassiveMode(INT iPassive);

引数	**戻り値**
iPassive：(In)0：通常モード 0 以外：パッシブモード ProEasy 初期化時は「通常モード」です。	正常終了：0 異常終了：エラーコード

特記事項

27.8 その他の API

関数名	GP 時間 DWORD 型読み出し

指定された局の現在時刻を数値（DWORD 型）として取得する関数です。ただし、LS2048 から 6 ワードに時刻が保存されているものに対してのみ有効な関数です。

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

引数
sNodeName：取得元の局名（Pro-Server EX 局は指定できません。）
odwTime：取得された時刻（時刻は DWORD 型（実体は ANSI で定義されている time_t 型）で取得）

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	GP 時間 VARIANT 型読み出し

指定された局の現在時刻を数値（Variant 型）として取得する関数です。ただし、LS2048 から 6 ワードに時刻が保存されているものに対してのみ有効な関数です。

DWORD WINAPI EasyGetGPTimeVariant(LPCSTR sNodeName, LPVARIANT ovTime);

引数
sNodeName：取得元の局名（Pro-Server EX 局は指定できません。）
ovTime：取得された時刻（取得する時刻は VARIANT 型（内部処理形式は Date）で取得）

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	GP 時間 STRING 型読み出し

指定された局の現在時刻を、文字列（LPTSTR 型）として取得する関数です。ただし、LS2048 から 6 ワードに時刻が保存されているものに対してのみ有効な関数です。

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

引数
sNodeName：取得元の局名（Pro-Server EX 局は指定できません）
pFormat：文字列として取得する時刻の書式設定文字列。パーセント記号（%）に続く書式指定コードが、＜特記事項＞に示すものに置き換えられます。
なお、それ以外の文字は、変更されないでそのまま表示されます。
osTime：文字列として取得した時刻（ただし、取得される文字列長＋ 1（NULL）以上の領域が確保されていない場合、予期せぬメモリ領域の破壊が発生します。そのため、予想される文字列長＋ 1（NULL）以上の領域を確保する必要があります。確保されていない場合、動作保証はいたしません。）

戻り値
正常終了：0
異常終了：エラーコード

特記事項
パーセント記号（%）に続く書式指定コードが、次表に示すものに置き換えられます。なお、それ以外の文字は、変更されないでそのまま表示されます。例えば、実際の時刻が 2006/1/2 12:34:56 の場合に「%Y_%M %S」と指定すると、文字列は「2006_34 56」となります。

書式指定コード	フォルダ
%a	曜日の省略名（※ 2）
%A	曜日の正式名（※ 2）
%b	月の省略名（※ 2）
%B	月の正式名（※ 2）
%c	ロケールに応じた日付と時間の表現
%#c	ロケールに応じた日付と時間の長い表現
%d	10 進数で表す月の日付（01 ～ 31）（※ 1）
%H	24 時間表記の時間（00 ～ 23）（※ 1）
%I	12 時間表記の時間（01 ～ 12）（※ 1）
%j	10 進数で表す年頭からの日数（001 ～ 366）（※ 1）
%m	10 進数で表す月（01 ～ 12）（※ 1）
%M	10 進数で表す分（00 ～ 59）（※ 1）
%p	現在のロケール AM/PM（※ 2）
%S	10 進数で表す秒（00 ～ 59）（※ 1）
%U	10 進数で表す週の通し番号。日曜日を週の最初に日とする（00 ～ 53）（※ 1）
%w	10 進数で表す曜日。日曜日を 0 とする（0 ～ 6）（※ 1）
%W	10 進数で表す週の通し番号。月曜日を週の最初の日とする（00 ～ 53）（※ 1）
%x	現在のロケールの日付表示
%#x	現在のロケールに応じた長い日付表示
%X	現在のロケールの時刻表示（※ 2）
%y	10 進数で表す西暦の下 2 桁（00 ～ 99）（※ 1）
%Y	10 進数で表す 4 桁の西暦（※ 1）
%z、%Z	時間帯の名前またはその省略名。時間帯がわからない場合には文字を入れない（※ 2）
%%	パーセント記号（※ 2）

※ 1：d、H、I、j、m、M、S、U、w、W、y、Y の前に # をつける（例えば、%#d）と、先行ゼロがあれば削除されます。（例えば、05 の場合は 5 となります。）

※ 2：a、A、b、B、p、X、z、Z、% の前に # をつける（例えば、%#a）と、# は無視されます。

関数名	GP 時間 STRING VARIANT 型読み出し

指定された局の現在時刻を文字列（Variant 型）として取得する関数です。ただし、LS2048 から 6 ワードに時刻が保存されているものに対してのみ有効な関数です。

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

引数
sNodeName：取得元の局名（Pro-Server EX 局は指定できません）
pFormat：文字列として取得する時刻の書式設定文字列。パーセント記号（%）に続く書式指定コードが、以下に示すものに置き換えられます。なお、それ以外の文字は、変更されないでそのまま表示されます。（詳細については、「GP 時間 STRING 型読み出し関数」の＜特記事項＞を参照してください。）
ovTime：文字列として取得した時刻（取得する時刻は VARIANT 型（内部処理形式は BSTR）で取得）

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	参加局ステータス読み出し

接続されている機器（GP）状態を取得することができます。また、レスポンスタイムアウト値を可変に設定できるため、接続確認にも使用できます。

シングル
INT WINAPI GetNodeProperty(LPCSTR sNodeName,DWORD dwTimeLimit,LPSTR osGPType,LPSTR osSystemVersion,LPSTR osComVersion,LPSTR osECOMVersion);
マルチ
INT WINAPI GetNodePropertyM(HANDLE hProServer,LPCSTR sNodeName,DWORD dwTimeLimit,LPSTR osGPType,LPSTR osSystemVersion,LPSTR osComVersion,LPSTR osECOMVersion);

引数
hProServer：（In）Pro-Server のハンドル
sNodeName：（In）読み出し先 GP の局名
dwTimeLimit：（In）レスポンスタイムアウト設定値
（0 を設定した場合はデフォルト値 3000msec）
設定範囲は 1 ～ 2,147,483,647、単位 msec

以下のエリアに対象局の情報が返されます。
それぞれ 32 バイト以上のエリアを確保してください。
osGPType：（Out）GP 機種コード
osSystemVersion：（Out）GP システムバージョン
osComVersion：（Out）PLC プロトコルドライバのバージョン
Pro-Server EX 局、GP3000 シリーズ局、WinGP 局および LT3000 局の場合は空白です。
osECOMVersion：（Out）2Way ドライバのバージョン
Pro-Server EX 局、GP3000 シリーズ局、WinGP 局および LT3000 局の場合は空白です。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

関数名	シンボル／グループのバイトサイズを求める

デバイスシンボルやグループシンボルにアクセスするために必要なバッファの合計バイト数を求めます。

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

引数
sNodeName：（In）機器名付き参加局名
sSymbolName：（In）求めたいデバイスシンボル名もしくはグループシンボル名
oiByteSize：（Out）求めたいバイトサイズ

戻り値
正常終了：0
異常終了：エラーコード

特記事項
sSymbolName にはデバイスシンボル、非配列グループ、配列グループ全体の全体、配列グループの 1 要素分を指定することができます。

関数名	グループのメンバー数を求める

指定されたグループシンボルもしくはシンボルシート内のメンバー（シンボルとグループの合計）数を求めます。

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

引数
sNodeName：（In）機器名付き参加局名
sSymbolName：（In）求めたいグループシンボル名またはシンボルシート名
oiCountOfMember：（Out）求めたいメンバー数

戻り値
正常終了：0
異常終了：エラーコード

特記事項
指定されたグループシンボル内にグループシンボルがある場合、内側のグループシンボルに複数のデバイスシンボルがいくつあっても、1 メンバーとカウントされます。

関数名	シンボル／グループ／シンボルシートの定義情報を求める

指定されたデバイスシンボル／グループシンボル／シンボルシートの定義情報（データ型やデータ数など）を求めます。

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

引数
sNodeName：（In）接続機器名付き参加局名
sSymbolName：（In）シンボル／グループ名／シート名
iMaxCountOfSymbolMember：（In）求める情報の最大数 1 以上の値を指定してください。用意した oSymbolInformation の個数を指定します。
osSymbolSheetName：（Out）sSymbolName が属しているシンボルシートの名前を返します。66 バイト以上のワークを用意してください。
oSymbolInformation：（Out）求めた詳細情報を配列の形で返します。iMaxCountOfSymbolMember で指定した個数分をワークとして用意してください。
oiGotCountOfSymbolMember：（Out）実際に oSymbolInformation に返した情報数を返します。

戻り値
正常終了：0
異常終了：エラーコード

特記事項

• SymbolInformation の構造

```
struct SymbolInformation
{
   WORD m_wAppKind;// データタイプ　シンボルの場合は 1 ～ 12,
グループの場合は 0x8000
   WORD m_wDataCount; // データ数
   DWORD m_dwSizeOf; // アクセスするのに必要なバッファのバイト数
   char m_sSymbolName[64+1];// シンボルもしくはグループの名前
   char m_bDummy1[3];// 予約
   char m_sDeviceAddress[256+1]; // デバイスアドレス （グループの場合は空白です )
   char m_bDummy2[3];// 予約
};
```

oSymbolInformation に求めた情報が、SymbolInformation 構造の配列で返りますが、1 個目には sSymbolName で指定されたシンボルもしくはグループ、シートの情報がセットされます。
2 個目以降には、sSymbolName がグループの場合は、グループのメンバーの情報がセットされます。
sSymbolName がシートの場合は、シート全体の情報がセットされます
sSymbolName がシンボルの場合には、2 個目以降はありません。

対象のシンボルがビットオフセットシンボルの場合、以下の点について注意してください。

①ビットオフセットシンボルを情報元のシンボルとして直接指定した場合（sSymbolName にビットオフセットシンボルを直接指定した場合）は、oSymbolInformation の 1 個目の SymbolInformation の m_dwSizeOf には、ビットシンボルをアクセスするためのバイト数 2 がセットされます。
情報元が 1 シンボルなので、oSymbolInformation の 2 個目以降はありません。

②情報元のシンボルとしてはグループシンボルを指定し、そのグループ内にビットオフセットシンボルがある場合、oSymbolInformation の 2 個目以降の m_dwSizeOf は、グループアクセス時のメンバーとしてアクセスサイズなので 0 がセットされます。

• メンバー数が不明な場合は、GetCountOfSymbolMember() でメンバー数を求め、その値＋ 1 の分の SymbolInformation をワークとして用意し、この関数を呼び出してください。

27.9 API を使用する場合の注意事項

■『Pro-Server EX』で利用可能なデータ型について

① API でデータタイプを指定する場合、または答えとして受け取る場合のデータタイプの基本形

定義名	10 進値	16 進値	データの意味
EASY_AppKind_Bit	1	0x0001	Bit Data
EASY_AppKind_SignedWord	2	0x0002	16Bit(Signed) Data
EASY_AppKind_UnsignedWord	3	0x0003	16Bit(Unsigned) Data
EASY_AppKind_HexWord	4	0x0004	16Bit(HEX) Data
EASY_AppKind_BCDWord	5	0x0005	16Bit(BCD) Data
EASY_AppKind_SignedDWord	6	0x0006	32Bit(Signed) Data
EASY_AppKind_UnsignedDWord	7	0x0007	32Bit(Unsigned) Data
EASY_AppKind_HexDWord	8	0x0008	32Bit(HEX) Data
EASY_AppKind_BCDDWord	9	0x0009	32Bit(BCD) Data
EASY_AppKind_Float	10	0xA	単精度浮動小数点データ
EASY_AppKind_Real	11	0xB	倍精度浮動小数点データ
EASY_AppKind_Str	12	0xC	文字列データ

② 特殊な場合に利用可能なデータタイプ

定義名	10 進値	16 進値	データの意味
EASY_AppKind_NULL	0	0x0000	デバイスアドレスとしてシンボルを利用可能な API でそのシンボルに定義づけされているデータ型をデータ型として利用することを示します。
EASY_AppKind_BOOL	513	0x0201	Bit データを 1 ビット単位で VARINT 型の BOOL として扱います。
EASY_AppKind_Group	-32768	0x8000	グループシンボル
EASY_AppKind_SymbolSheet	-28672	0x9000	シンボルシート

■ 接続機器名付き参加局名について

① GP3000 シリーズ局、WinGP 局および LT3000 局は複数の機器と接続することができます。それらのデバイスへアクセスする場合は、局名と接続機器名まで指定する必要があります。

② Pro-Server EX API の引数には参加局名までの指定でよいものと、接続機器名まで指定しなければならないものがあります。

＜接続機器名の指定のしかた＞

参加局名に続き“ . ”（ドット）を付加し、接続機器名を指定します

例）

AGPNode.PLC1

③ GP3000 シリーズ局、WinGP 局、LT3000 局、Pro-Server EX 局の内部デバイスへアクセスする場合、接続機器名には “ #INTERNAL ” と指定してください。（ただし、省略は可能です。）

④ GP3000 シリーズ局、WinGP 局および LT3000 局のメモリリンクドライバを利用していて、そのドライバーのメモリへアクセスする場合、接続機器名に“ #MEMLINK ” と指定してください。（省略できません。）

⑤ GP シリーズ局、Pro-Server EX 局の場合は、接続機器名の指定は必要ありません。

. (ドット) も必要ありません。

⑥ GP3000 シリーズ局、WinGP 局および LT3000 局の内部デバイスまたは“ システムエリア機器 ” に割り付けられている接続機器は、接続機器名付き参加局名の指定で、接続機器名を省略することができます。

ただしその場合、対象デバイスについて『Pro-Server EX』は先に内部デバイスを探し、なければ“ システムエリア機器 “ に割り付けられている接続機器を探します

■ シンボルの検索順について

『Pro-Server EX』の API デバイスへアクセスする API は、接続機器名付き参加局名とデバイスアドレスまたはデバイスシンボルを文字列で指定しますが、『Pro-Server EX』は以下の順位で指定された文字列がデバイスアドレスを直接示す文字列なのか、デバイスシンボルなのかを判断します。

①シンボルシート名と合致するか探します。その中にあれば、シートとして扱います。

②指定された文字列をグループ名もしくはシンボルとみなし、ローカルシンボルシートの中から探します。その中にあれば、ローカルシンボルと見なします。

③ローカルシンボルになければグローバルシンボルシートの中から探します。（このとき、検索の対象となるグローバルシンボルは、指定された“ 接続機器名付き参加局名 ” と同じ機器のグローバルシンボルシートです。異なる機器のグローバルシンボルシートは対象になりません。）

④なければデバイスアドレスとみなします。

■ 名前の重複

『Pro-Server EX』は、名前として以下の種類があります。
①参加局名
②接続機器名
③起動条件名
④シンボルシート名
⑤グループ／シンボル名
⑥アクション名

『Pro-Server EX』では、名前は基本的に重複してはいけません。
ただし、以下の場合は例外となります。
①参加局が違う場合、接続機器名は重複しても問題ありません。
②参加局、接続機器が違う場合、グループ／シンボル名は重複しても問題ありません。

■ グローバルシンボルとローカルシンボルで名前が重複している場合

『Pro-Server EX』の API でデバイスアドレスにシンボルを利用する場合で、そのシンボルがローカルシンボルとグローバルシンボルの両方にある場合（名前が重複している場合）は、ローカルシンボルとして処理されます。

■ マルチスレッドのアプリケーションで Pro-Server EX API を使用する場合

Pro-Server EX API の関数はすべて同期型（関数を呼び出すと処理が完了するまで関数から復帰しないタイプ）です。
そのため、複数の参加局に対しアクセスする場合は、単一スレッドのプログラムでは 1 局ずつ順番に処理することになります。
マルチスレッドのプログラムでは、1 つのスレッドが 1 つの参加局にアクセス中でも、別のスレッドから別の参加局にアクセスできます。
Pro-Server EX API はマルチスレッドに対応しています。
以下にマルチスレッドのプログラムを組む場合に気を付けていただきたい点について説明します。

①マルチスレッドのプログラムは、基本的にはマルチハンドル系の関数をご利用ください。

②マルチハンドル系の関数を利用するには、『Pro-Server EX』のハンドルを取得する必要がありますが、スレッドごとに別々の『Pro-Server EX』のハンドルを取得し使用してください。
1 スレッド内で複数の『Pro-Server EX』のハンドルを取得し使用しても問題ありませんが、別のスレッドで作成した『Pro-Server EX』のハンドルを使用しないでください。
同様に、『Pro-Server EX』のハンドルを開放する場合は、そのハンドルを作ったスレッドで行ってください。

③ Pro-Server EX API を使用する場合、最初に EasyInit() を呼び出す必要があります。
しかし、ほとんどの『Pro-Server EX』の API は、EasyInit() を呼び出す前に呼ばれた場合、内部で自動的に EasyInit() を呼び出しています。
それにより、シングルスレッドの場合、EasyInit() を意識する（プログラム化する）必要はありません。

④ EasyInit() を呼び出したスレッドはアプリケーションが終了する最後まで存在していなければなりません。途中で、EasyInit() を呼び出したスレッドが終了した場合、動作は保証できません。

⑤通常のアプリケーションは、アプリケーション起動時のスレッドが、最後まで存在していることが多い（VB や VC で作成したアプリケーションは通常そのようになります）ため、マルチスレッドのアプリケーションを構築する場合は、アプリケーションの最初に EasyInit() をコールすることをおすすめします。

■ 効率よくキャッシュを更新するには

①キャッシュ機能を利用するには、キャッシュバッファにデバイスを登録する必要があります。
（『Pro-Studio EX』のキャッシュ登録画面で登録するか、キャッシュバッファ制御 API で登録します。）
このとき、登録のしかたにより、システム全体のパフォーマンスが違います。

②どのデバイスを登録するかは、デバイスアクセスログ機能を利用すると、『Pro-Server EX』がどのデバイスにアクセスしているかが分かります。

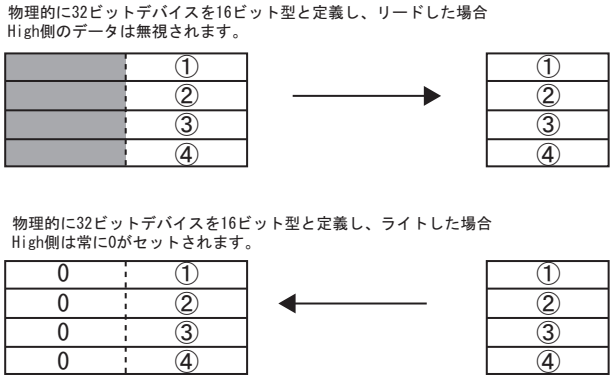
③基本的には、ひんぱんにリードしているデバイスをキャッシュ登録します。

④複数のデバイスを登録する場合、できる限り 1 連続で登録するほうが高速です。
（例 1） LS100 と LS101 をキャッシュ登録する場合、別々に登録するより、LS100 から 2 個登録する方が高速になります。さらに、2 つのデバイスの間が数ワード程度の場合、1 連続として登録した方が高速な場合があります。
（例 2） LS100 と LS103 をキャッシュ登録する場合、別々に登録するより、LS100 から 4 個登録する方が高速になります。

⑤連続するビットデバイスを登録する場合、可能であればワードデバイスとして登録するほうが高速になります。
（例） LS123401 から 20 ビット分連続して登録する場合、LS1234 から 2 ワード登録するほうが高速になります。

■ 物理的に 32 ビット幅のデバイスに対し 16 ビットアクセスした場合の動作について

①『Pro-Server EX』では、物理的に 32 ビット幅のデバイスに対し、16 ビットのシンボルを割り付け、そのシンボルでアクセスした場合や、データ型を直接 16 ビット型としてアクセスした場合など、32 ビット幅のデバイスに対し 16 ビットデバイスとして扱うことができます。
その場合、リードとライトで以下のような変換が行われます。



②この変換はデータ転送機能、API を使用したアクセス時に行われます。

③ただし、GP シリーズ局←→ GP シリーズ局でのデータ転送ではエラーになります。

④旧 Pro-Server では、物理的に 32 ビット幅のデバイスに対し、16 ビットアクセスを行うとエラーになります。

■ 物理的に 16 ビット幅のデバイスに対し 32 ビットアクセスした場合の動作について

『Pro-Server EX』では、物理的に 16 ビット幅のデバイスに対し、32 ビットのシンボルを割り付け、そのシンボルでアクセスした場合や、データ型を直接 32 ビット型としてアクセスした場合など、16 ビット幅のデバイスに対し 32 ビットデバイスとして扱うことができます。
その場合、連続する 2 つの 16 ビット幅のデバイスを 1 デバイスとして扱います。

■ Pro-Server の自動起動と途中終了、再開について

① Pro-Server EX API は、『Pro-Server EX』を一度も起動していなければ、自動的に起動します。
（一部 API を除きます。）もし、『Pro-Server EX』を起動できなければ、以降 API は常にエラーを返します。

②『Pro-Server EX』が正常に起動したあと、2 回目以降の API コールについては、『Pro-Server EX』は既に起動しているので、多重起動されることはありません。

③アプリケーションの処理の途中で『Pro-Server EX』を終了させ、その後 API をコールした場合（2 回目以降の API コールで『Pro-Server EX』が終了している場合）、API は『Pro-Server EX』を起動しません。API はエラーになります。

④アプリケーションの処理の途中で『Pro-Server EX』を終了させないでください。
『Pro-Server EX』を終了させる場合は、必ず先にアプリケーションを終了させてください。（Pro-Server EX 終了後に API をコールしないでください）
ただし、『Pro-Server EX』を Windows のスタートメニューなどから手動で再起動した場合、API は『Pro-Server EX』の復旧処理を行い、処理の継続を試みます。復旧できれば処理を継続できます。ただし、前回の Pro-Server EX の終了の仕方によっては復旧処理が失敗することがあります
復旧処理が失敗する事例として以下のような場合があります
・『Pro-Server EX』をタスクマネージャなどから強制的に終了させた場合。
・API コール中に『Pro-Server EX』を終了させた場合

■ シンボルのインデックス指定について

API のデバイス名でのみ、シンボルのインデックス指定ができます。シンボルのインデックス指定とは下記のとおり、シンボル名の後ろに [] で数値を指定する方法です。意味は、そのシンボルのデータ型で「数値」で指定されている個数分、シンボル名で表されるデバイスから進めたデバイスを指します。

(シンボル名)[数値]

例）バルブ [2]
バルブというシンボルが D100 で 16 ビット符号ありと指定されていた場合、D102 を指すことになります。また、D100 で 32 ビット符号なしと指定されていた場合は、D104 を指すことになります。

■ キューイングやシンボルのキャッシュリードについて

キューイングのキャッシュリード（BeginQueuingRead 後、ReadDevice 系関数（D なし）でキューイング登録）や、シンボルのキャッシュリード (ReadSymbol(D なし)) を使用する場合は、読み込み対象のデバイスのどの部分がキャッシュ登録されているかによって動作が変わるのでご注意ください。

- 読み込み対象のデバイスすべてがキャッシュ登録されている場合：キャッシュリードとなります。
- 読み込み対象のデバイスすべてがキャッシュ登録されていない場合：ダイレクトリードとなります。
- 読み込み対象のデバイスの一部のみがキャッシュ登録されている場合：一部のデバイスがキャッシュリードされ、残りのデバイスがダイレクトリードされます。ただし、キャッシュ登録されているデバイスがすべてキャッシュリードされるわけではなく、キャッシュ登録しているデバイスに対してもダイレクトリードされる場合があります。なお、このようにどのデバイスがキャッシュリードされるかわからないと困る場合は、読み込み対象のデバイスすべてをキャッシュ登録するか、キャッシュリードを使用せずにダイレクトリードを使用するように変更してください。

■ .NET で使用できない API について

.NET では次の API を使用することはできません。使用しても動作保証はいたしません。

- シンボルアクセス（Byte アクセス）

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

- シンボルサイズ取得関数

SizeOfSymbol()

■ ProEasy.h を VC でコンパイルすると LPVARIANT が未定義エラーになる場合

Visual C++ Ver.6 で PRO-SDK¥VC¥Public¥ProEasy.h または Pro-Studio の [プログラミング補助]-[VC：宣言文] でクリップボード経由で作成したヘッダをコンパイルすると LPVARIANT が未定義のエラーになることがあります。LPVARIANT は afxdisp.h の中で定義されていますので、これを include していないと未定義エラーになります。これを回避するには、通常は stdafx.h の中で #include <afxdisp.h> と定義するようにしてください。

■ マルチスレッドのアプリケーションで Pro-Server EX API を使用する場合

Pro-Server EX API は全て同期型（関数を呼び出すと処理が完了するまで関数から復帰しないタイプ）です。そのため、複数の参加局に対しアクセスする場合は、１スレッドのプログラムでは１局ずつ順番に処理します。

マルチスレッドのプログラムでは、１つのスレッドが１つの参加局にアクセス中でも、別のスレッドから別の参加局にアクセスできます。

Pro-Server EX API はマルチスレッドに対応しています。

以下にマルチスレッドのプログラムを組む場合に気を付けていただきたい点について説明します。

1. マルチスレッドのプログラムは基本的にはマルチハンドル系の関数をご利用ください。
2. マルチハンドル系の関数を利用するには Pro-Server EX のハンドルを取得する必要がありますが、レッド毎に別々の Pro-Server EX のハンドルを取得し利用してください。１スレッド内で複数の Pro-Server EX のハンドルを取得し利用するのは問題ありませんが、別のスレッドで作成した Pro-Server EX のハンドルを利用しないでください。同様に、Pro-Server EX のハンドルを開放する場合は、そのハンドルを作ったスレッドで行ってください。
3. Pro-Server EX API を利用する場合、最初に EasyInit() を呼び出す必要があります。
   ほとんどの Pro-Server EX API は、EasyInit() を呼び出す前に呼ばれた場合、内部で自動的に EasyInit() を呼び出しているので、EasyInit() の呼び出しを意識する（プログラム化する）必要はありません。
4. マルチスレッドのプログラムの場合、プログラムは最初に起動されたスレッド（メインスレッド）から最初に EasyInit() を呼び出す必要があります。
   メインスレッド以外のスレッドから Pro-Server EX の API をコールする場合は、その前にメインスレッドから EasyInit() をコールしてください。

■ Windows のメッセージ処理について

多くの Windows のプログラムは、「アイコンがクリックされた」、「マウスが動かされた」、「キーが押された」などのイベントに応じて、ダイアログを表示したり、音をならしたりする、イベント駆動型（イベントドリブン）のプログラムになっています。

Windows はイベントが発生すると、そのイベントの種類を示すメッセージをアプリケーションに送ります。

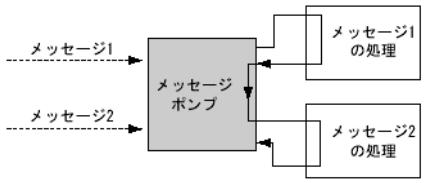
アプリケーションはメッセージを受け取ることでイベントが発生した事を確認し、それぞれの処理を行います。

本書では Windows からメッセージを順番に受け取り、個々の処理へ分岐する部分（VB なら DoEvents, に相当し、VC なら GetMessage() と DispatchMessage() を行う部分）をメッセージポンプと呼ぶことにします。

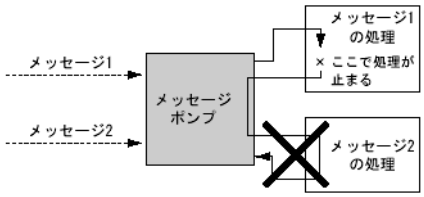
メッセージポンプは VC や VB で通常プログラムすると、VC や VB のフレームワークに隠蔽されているので、ほとんど気にしないと思いますが、このメッセージポンプが上手く動作しないと、Windows のアプリケーションは意図しない動作を行うアプリケーションになります。

例えば、あるメッセージを処理するルーチンが処理に時間がかかり、なかなか復帰しない場合、その間に発生したイベントをアプリケーションは Windows から受け取ることができないため、そのイベントの処理ができません。

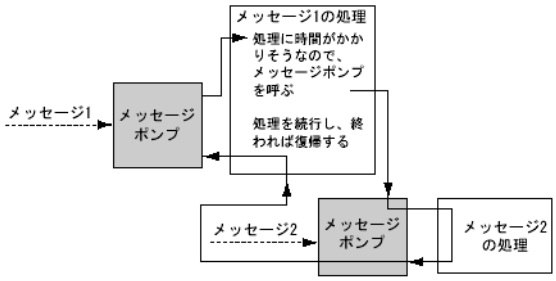
例）メッセージ１、メッセージ２の順番でメッセージが Windows から送られるとします。
メッセージポンプはメッセージ１を取り出し、メッセージ１用のサブルーチンを呼び出します。
そして、そこから復帰してくると次のメッセージ（メッセージ２）を取り出しメッセージ２用のサブルーチンを呼び出します。



この時、メッセージ１の処理が長い時間かかるとします。
すると、メッセージポンプへ復帰できないので、メッセージ２の処理ができません。



このような場合に、メッセージポンプを強制的に動かしてください。（VB なら DoEvents、VC なら GetMessage() と DispatchMessage() を呼ぶ）



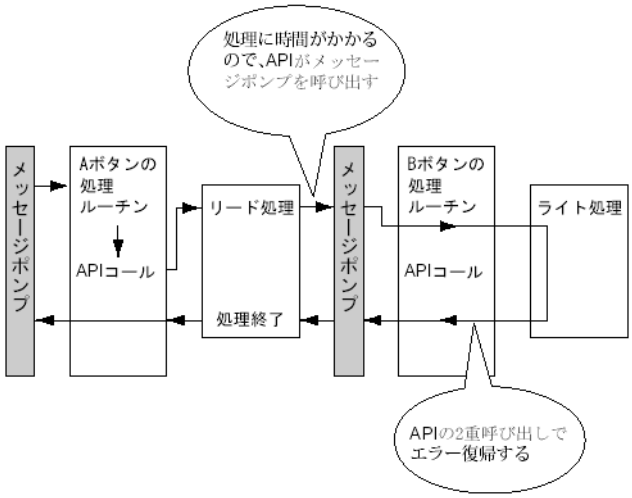
Windows のアプリケーションはアプリケーションが上手くメッセージポンプを動かす事を前提に作られた OS です。Pro-Server EX API は（例）に示したような事が起こらないように、時間のかかる処理の場合、関数内でメッセージポンプを動かしています。

■ API の二重呼び出しの禁止

Pro-Server EX API は、ある相手と通信中（Pro-Server EX の関数呼び出し中）にさらに、別の通信をすることを（二重呼び出し）禁止しています。（マルチハンドルを利用すれば可能です。詳しくはマルチハンドルの章を参照してください）しかし、Pro-Server EX API は、API 内でメッセージポンプを動かしていますので、イベントが発生すればユーザープログラムが動き出します。
メッセージの処理ルーチンの中で、API を呼び出すと二重呼び出しが発生することがあります。
二重呼び出しになる事例を以下に示します。

1.2 つのボタンを押すことによる二重呼び出し

2 つのボタン A と B があり、A が押されるとデバイスのリード API を呼び出し、B が押されるとデバイスのライト API を呼び出すとします。
この場合、A のボタンを押しデバイスのリード API を呼び出している最中に B のボタンを押すとデバイスのライト API が呼ばれ、その時点で API の二重呼び出しとなり、エラーとなります。



2.タイマーによる二重呼び出し

Windows のプログラムで周期的な処理を行う場合、よくタイマーイベントを利用しますが、タイマーイベントを利用するプログラムでは、気を付けてプログラムしないと、API を二重に呼び出してしまうことがあります。

① 1 秒に一回、周期的にデバイスリード API を呼び出しデバイスをリードし、それを表示する
② あるボタンが押されるとデバイスライト API を呼び出し、デバイスに値を書き込む

このようなプログラムは、以下のようなタイミングの場合、エラーになります。

・①のタイマーイベントが発生してリード中に、②のボタンを押し、②の処理が動き出したとき
・②のライト中にタイマーイベントが発生し、①のリードを行うとき

■ API の二重呼び出しの回避策

API の二重呼び出しの回避策には、以下のような方法があります。

① ユーザープログラムで API の二重呼び出しを行わないように、アルゴリズムを改良する

例えば、

１．タイマー処理ルーチンおよびボタンの処理ルーチンの先頭で、必ずタイマーのキャンセルを行う。

２．１つのボタンが押されて処理をしている間は、そのボタンや別のボタンが押されても無視するようにする。

② マルチハンドルを利用する

Pro-Server EX のハンドルが違えば API の二重呼び出しにはなりません。

マルチハンドルタイプの API を利用し、二重呼び出しの可能性のある部分のプログラムのハンドルを、別のハンドルにします。

③ API 内でメッセージ処理をしないようにする

EasySetWaitType() を引数 2 でコールする、ただし、この場合二重呼び出しの元となるメッセージ以外もメッセージについて処理されないので、アプリケーションが意図しない動作を行うなど他の問題が発生する事があります。

■ VB での文字列をリードする方法

VB で文字列をリードするには以下のような、2 種類の方法があります。

①VB で ReadDeviceStr を利用し文字列をリードする場合

この場合、事前にリードした文字列の格納先のサイズを指定（固定）する必要があります。

```
'
Public Sub Sample1()
    Dim strData As String * 10  ' 読み込むサイズを指定しているので正しい指定方法
    'Dim strData As String       ' 文字列のサイズを指定していないので誤った指定方法

    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
```

②VB で文字列を ReadDeviceVariant を利用しリードする場合
　事前にリードした文字列の格納先のサイズを指定しない場合は Variant 型を利用します。

```
'
Public Sub Sample2()

   Dim lErr As Long
   Dim vrData As Variant        ' 読込んだデータの格納先に Variant 型を指定します
   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
```

ここで気を付ける点として、GP では文字列の完結に NULL を利用しています。そのため、上記の方法で取得した文字列に文字列完結の NULL があれば、文字列を縮める必要があります。
文字列を NULL まで縮めるサンプル関数を示します。

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

27.10　API の使用例

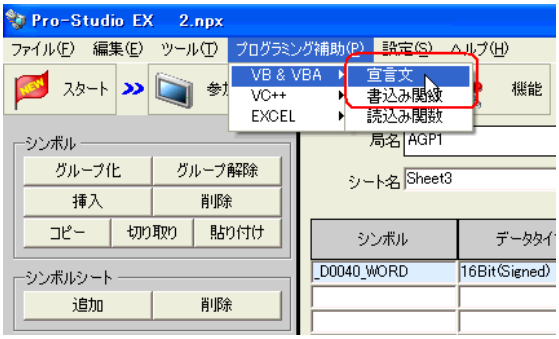
『Pro-Server EX』が提供する読み出し関数／書き込み関数を利用すると、ユーザーアプリケーションからデータの読み書きができます。
ここでは、API を使用して、指定シンボルを読み書きする手順について説明します。

☞「27.10.1　VB 機能補助」
☞「27.10.2　VC 機能補助」
☞「27.10.3　VB .NET 機能補助」
☞「27.10.4　C# .NET 機能補助」

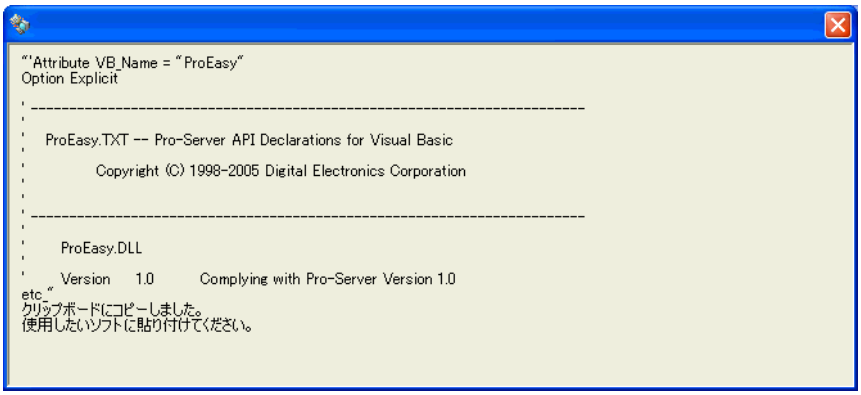
27.10.1　VB 機能補助

VB：宣言文

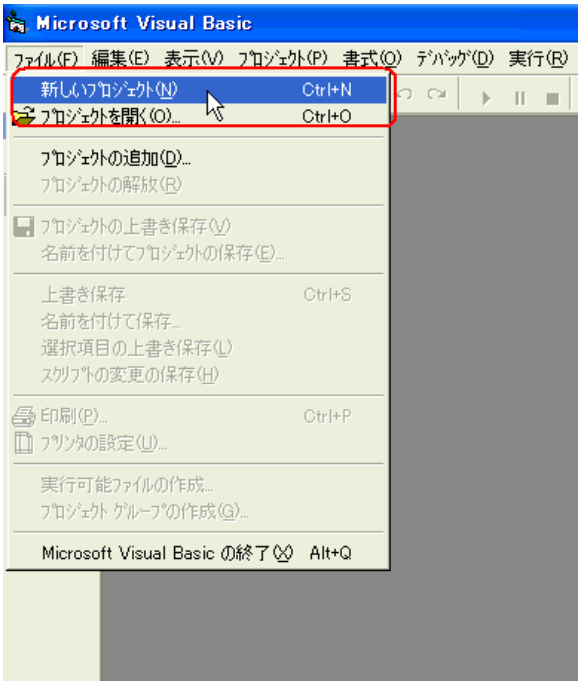
1［プログラミング補助］から［VB & VBA］→［宣言文］を選択します。



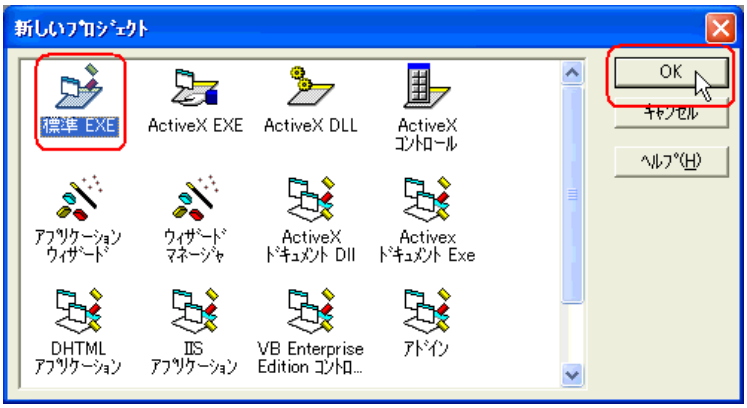
VB の宣言文がクリップボードにコピーされます。



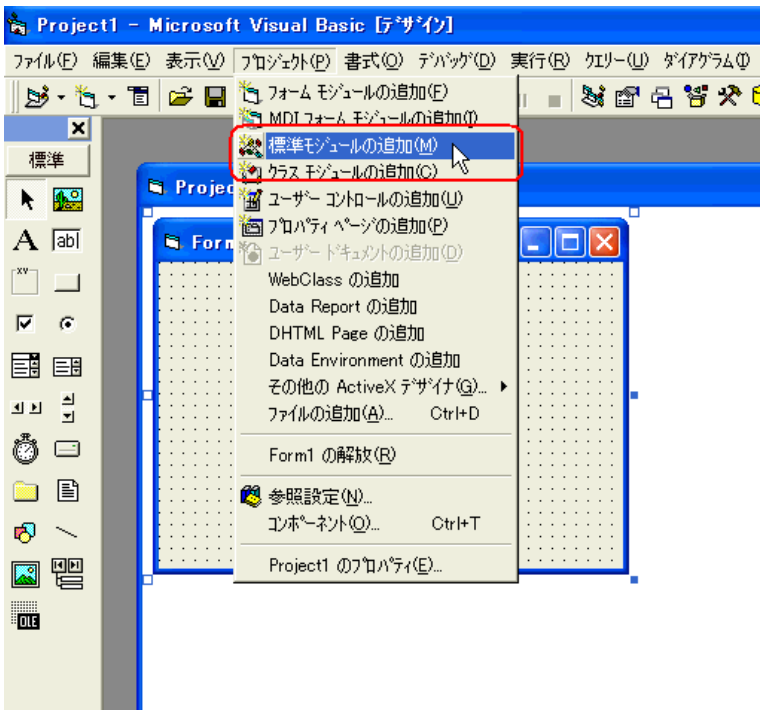
2 Microsoft Visual Basic を起動し、メニューの［ファイル］から［新しいプロジェクト］を選択します。



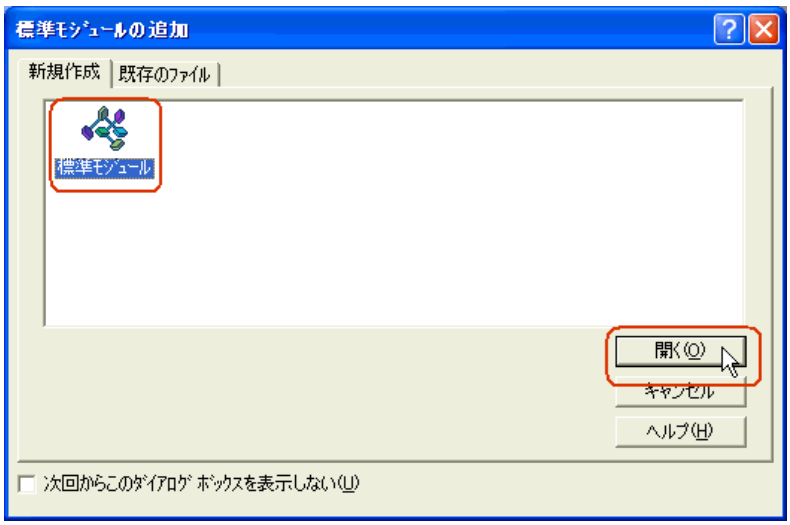
3［標準 EXE］を選択し、［OK］ボタンをクリックします。



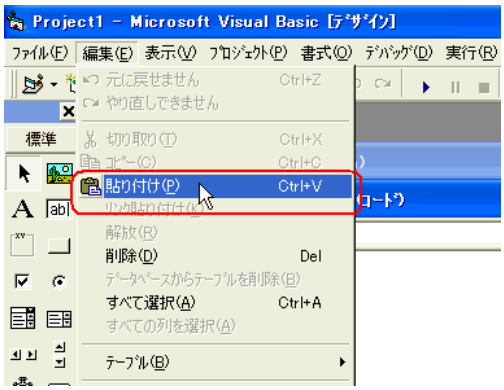
4 Microsoft Visual Basic のメニューの［プロジェクト］から［標準モジュールの追加］を選択します。



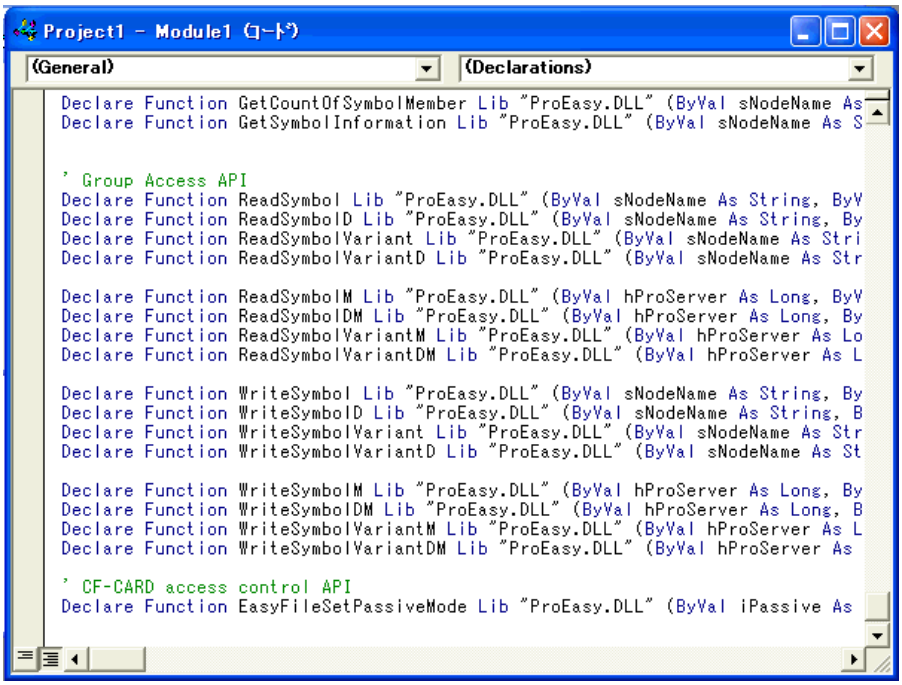
5［新規作成］タブの［標準モジュール］を選択し、［開く］ボタンをクリックします。



6 Microsoft Visual Basic のメニューの［編集］から［貼り付け］を選択し、追加された標準モジュールに宣言文（クリップボードの内容）を貼り付けます。



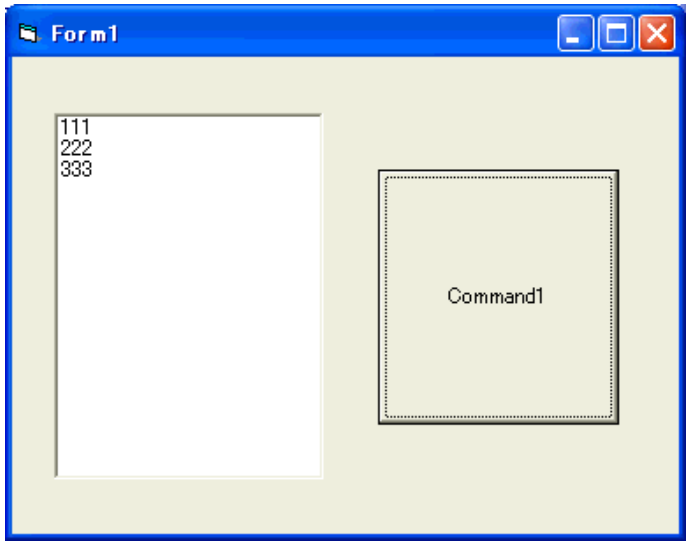
宣言文が貼り付けられます。



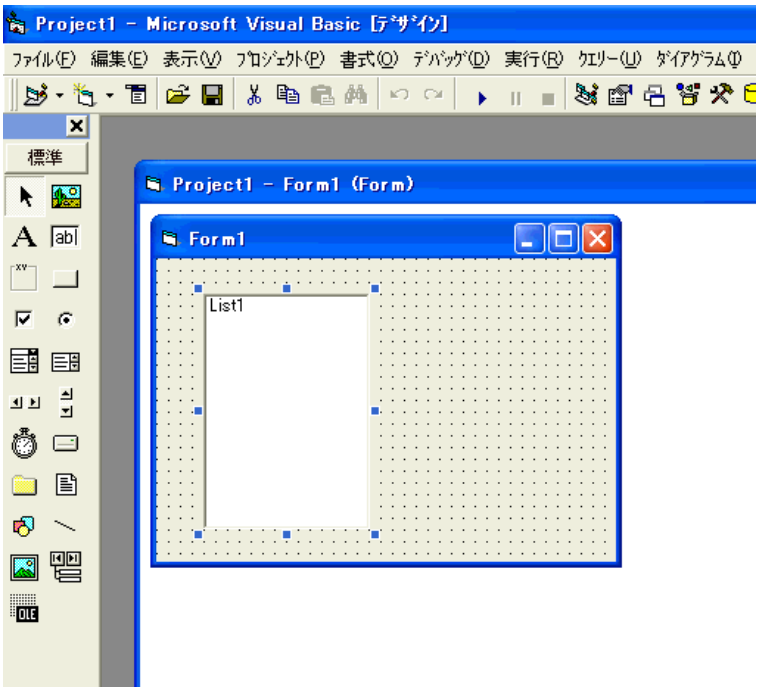
以上で、関数（読み込み関数 / 書き出し関数）の宣言は終了です。
前記 1 ～ 6 の操作は、読み込み／書き込みのいずれの場合でも共通です。
以降の手順については、読み込みの場合と書き込みの場合で手順が異なりますので、個別に説明します。
［読み込み］用アプリケーションの作成については、手順 7 ～ 16 をご覧ください。
［書き込み］用アプリケーションの作成については、手順 17 ～ 26 をご覧ください。

〔読み込み〕用アプリケーションの作成

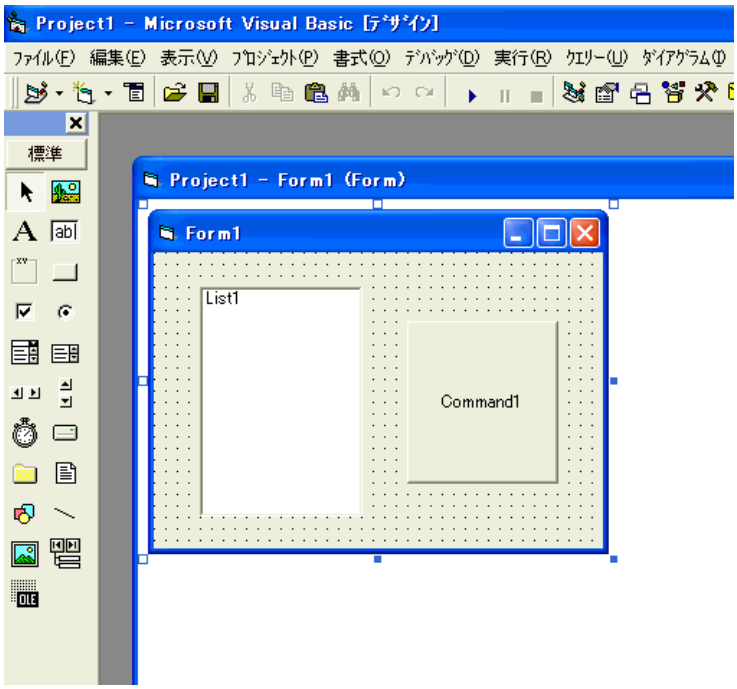
ここでは、［Command1］をクリックすると、3 点分のデータ（16 ビット符号付き）を読み出して表示するアプリケーションについて説明します。



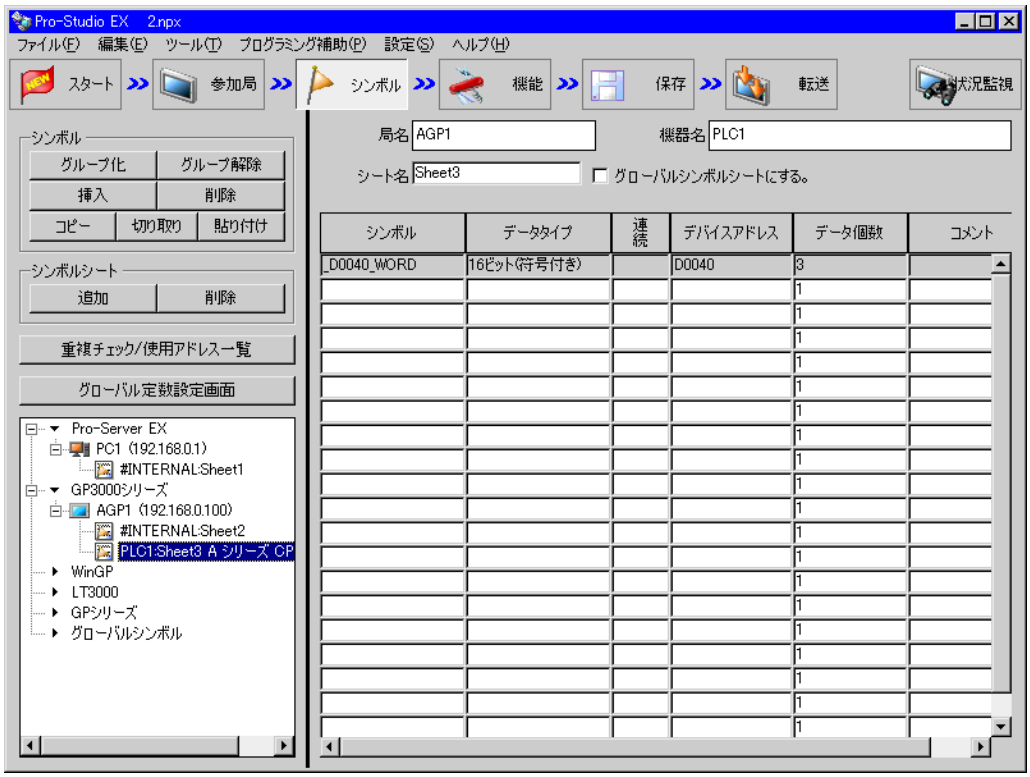
7［ListBox］を選択し、［Form1］に貼り付けます。



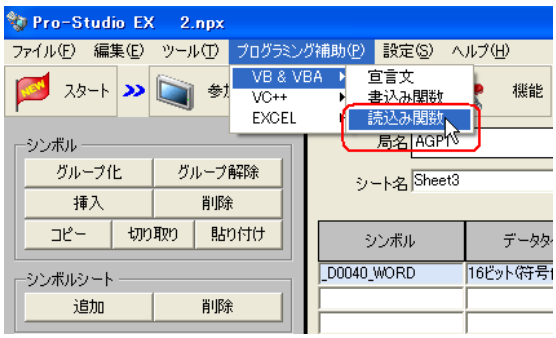
8［CommandButton］を選択し、［Form1］に貼り付けます。



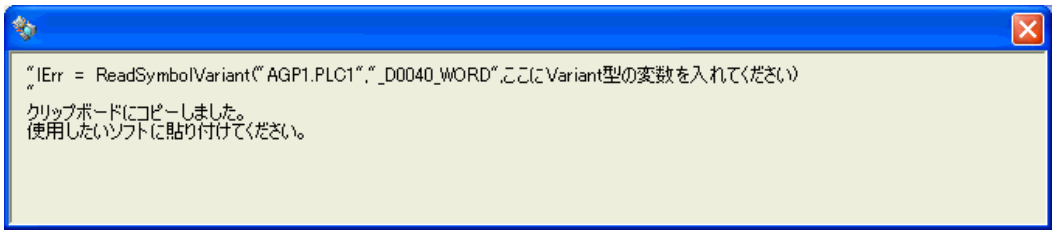
9『Pro-Studio EX』に登録されているシンボルで、読み込みたいシンボル名を選択します。（読み込み先の先頭を選択してください。）



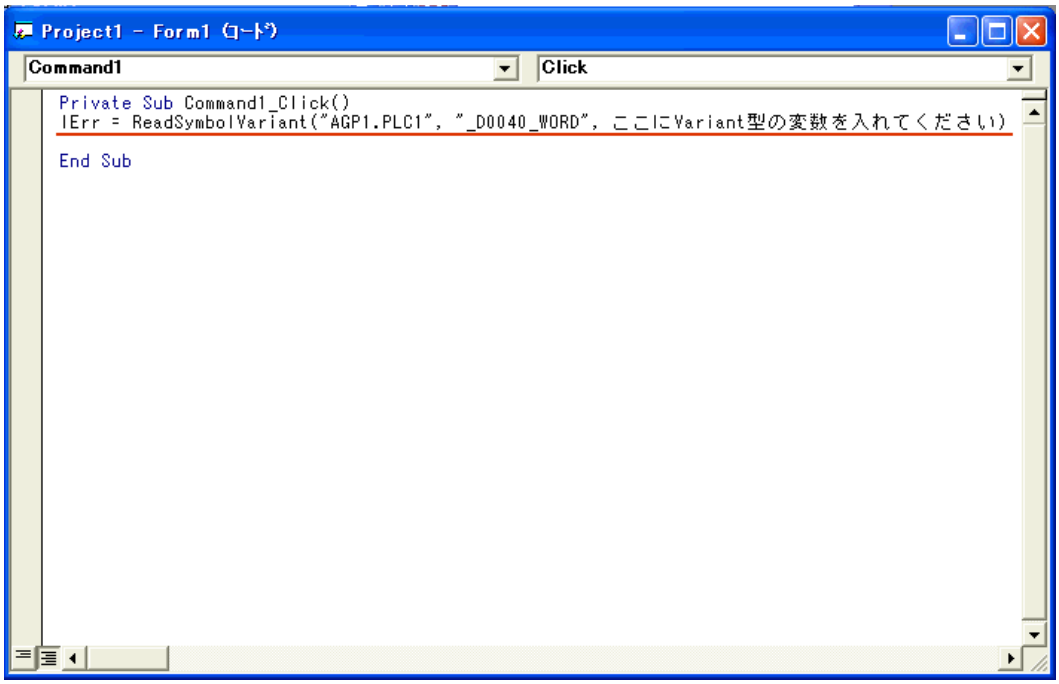
10 メニューの［プログラミング補助］から［VB & VBA］→［読込み関数］を選択します。



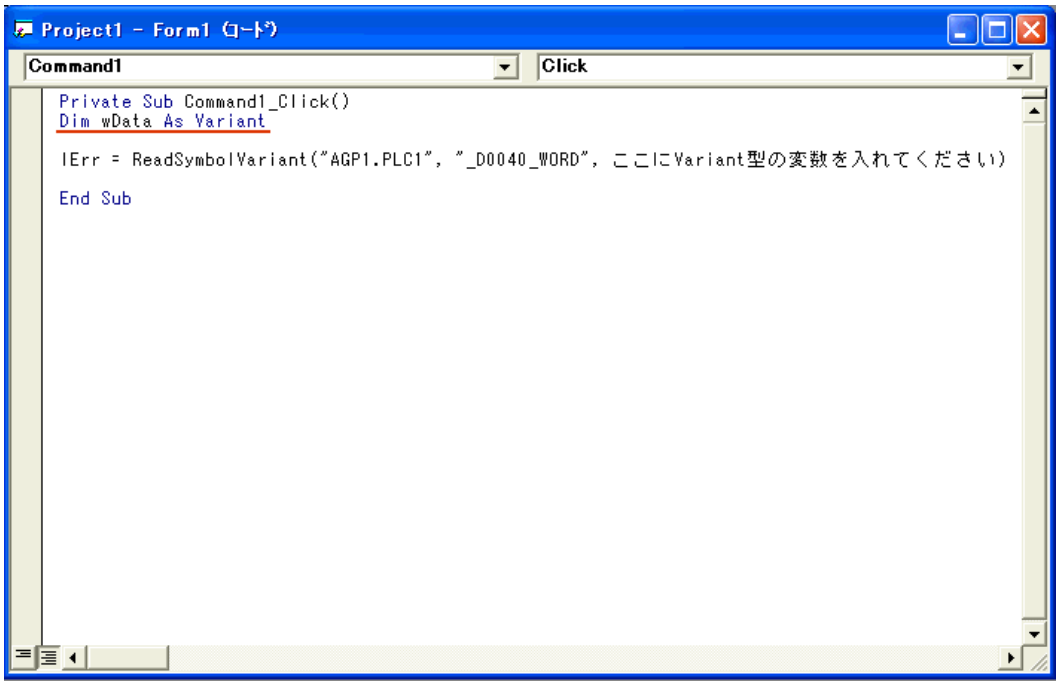
読み込み関数がクリップボードにコピーされます。



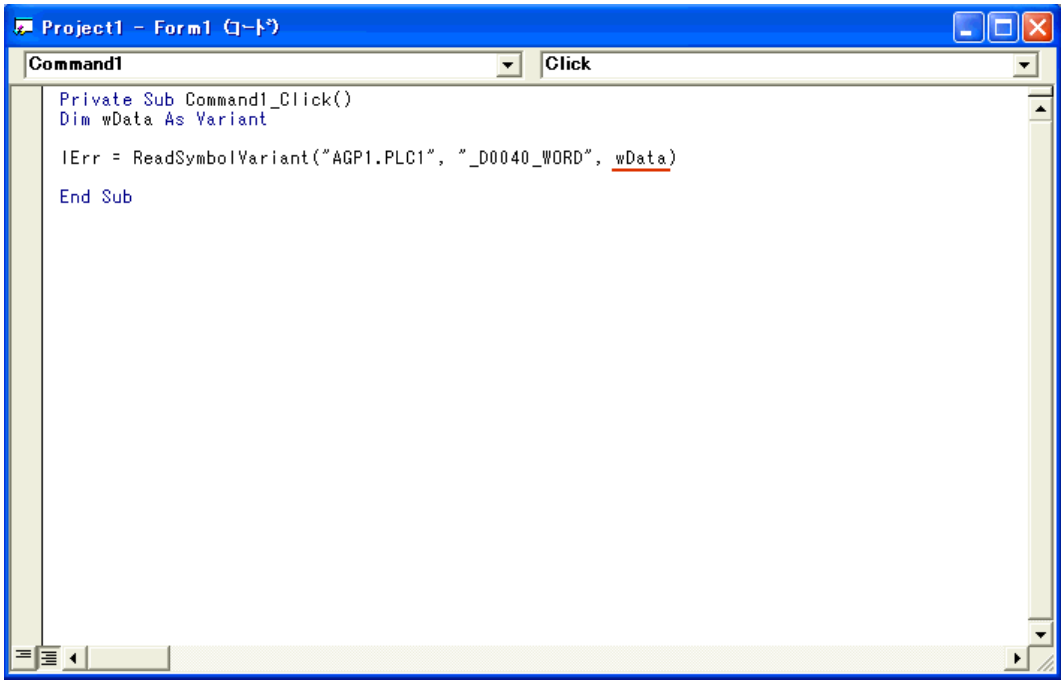
11 ［Form1］上の［Command1］をダブルクリックし、Sub ステートメントと End Sub ステートメントの間にクリップボードの内容（読み込み関数）を貼り付けます。



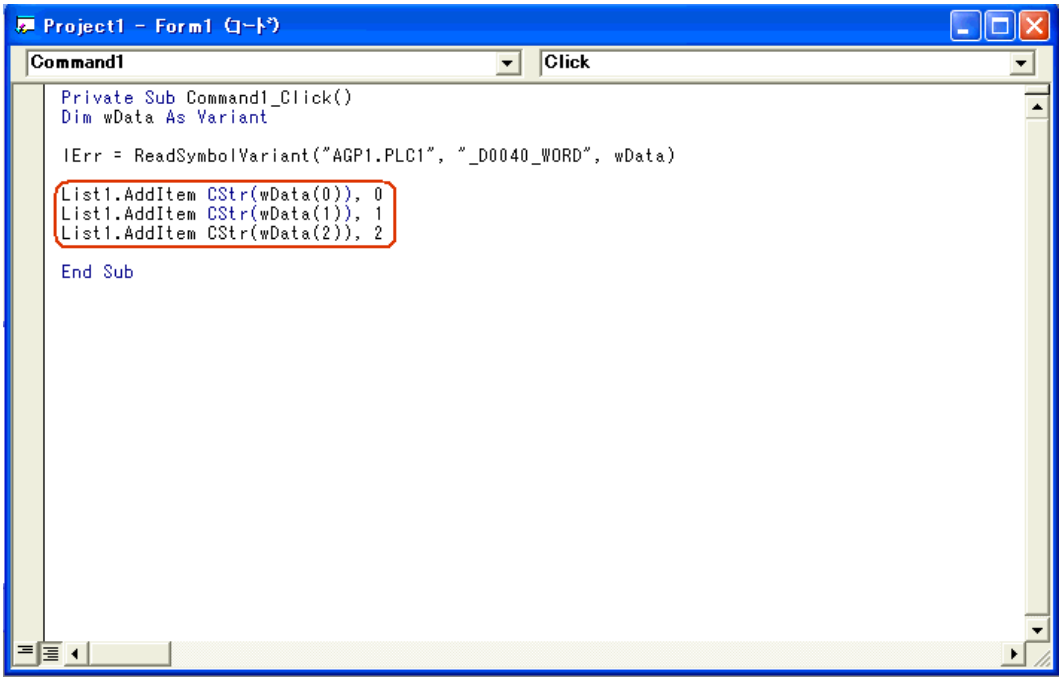
12 読み込んだデータを格納するエリア（配列）を宣言します。配列の型（本例では Variant）は、使用するシンボルのデータタイプに合わせてください。



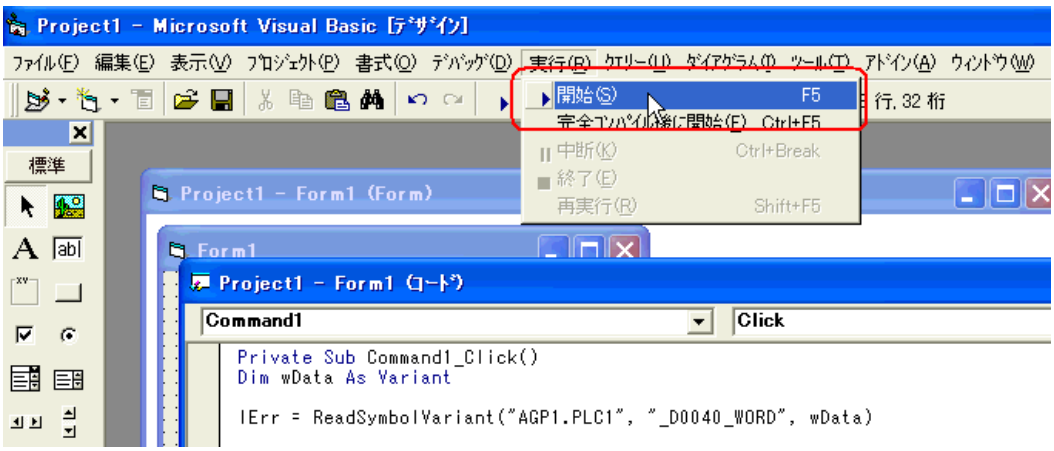
13 読み込んだデータを格納する先頭エリア（wData）を指定します。



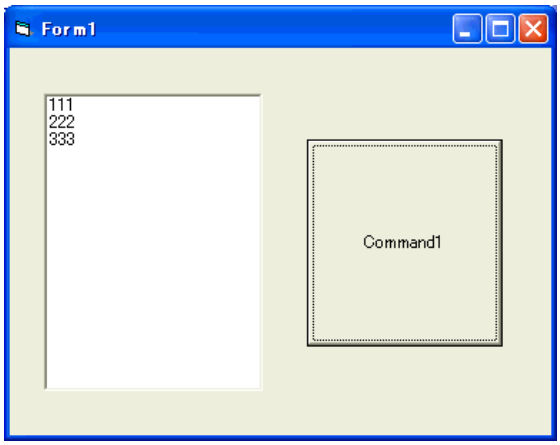
14 読み込んだデータ 3 点分（wData(0)、wData(1)、wData(2)）を ListBox に順次表示します。



15 Microsoft Visual Easic のメニューの［実行］から［開始］を選択します。

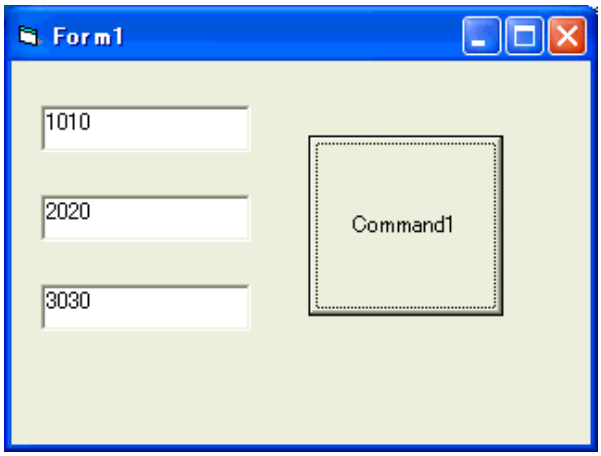


16 ［Command1］をクリックすると、シンボル“ _D0040_WORD ”から 3 点分のデータが ListBox に表示されます。

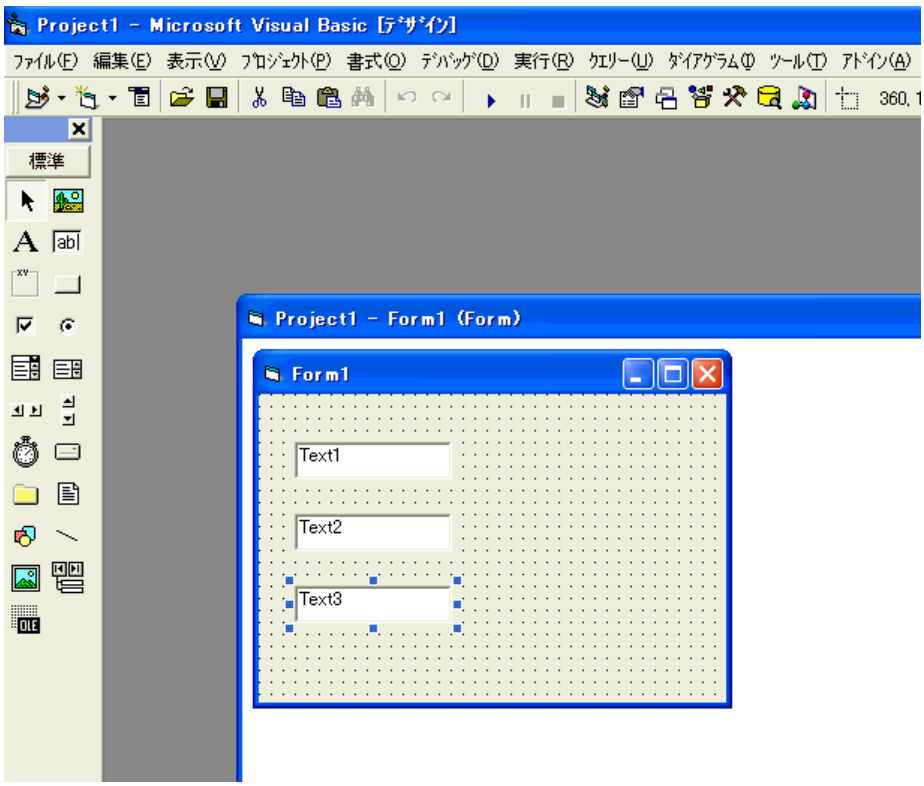


〔書き込み〕用アプリケーションの作成

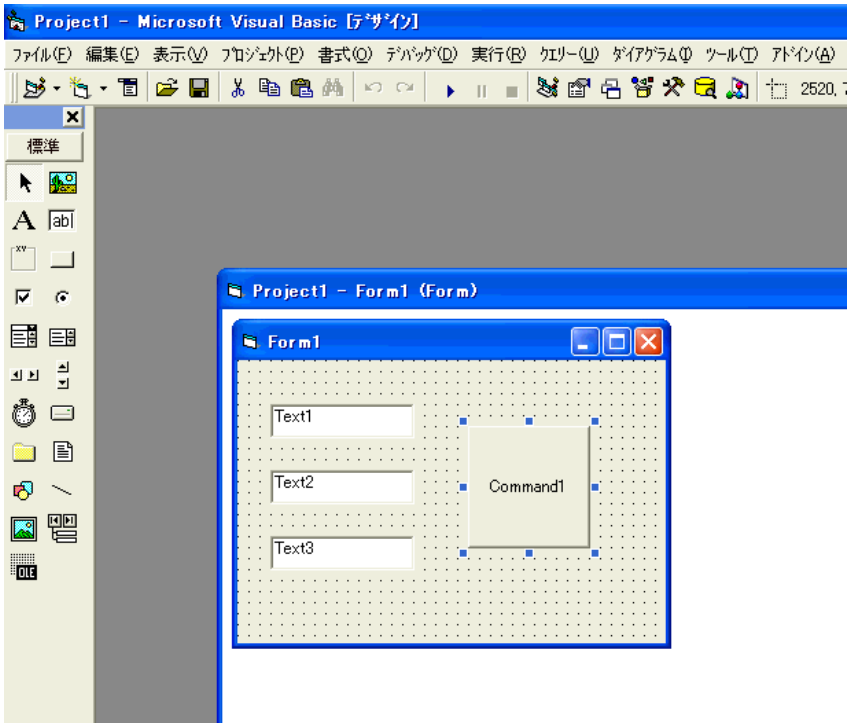
ここでは、［Command1］をクリックすると、入力された 3 点分のデータ（16 ビット符号付き）を書き込むアプリケーションについて説明します。



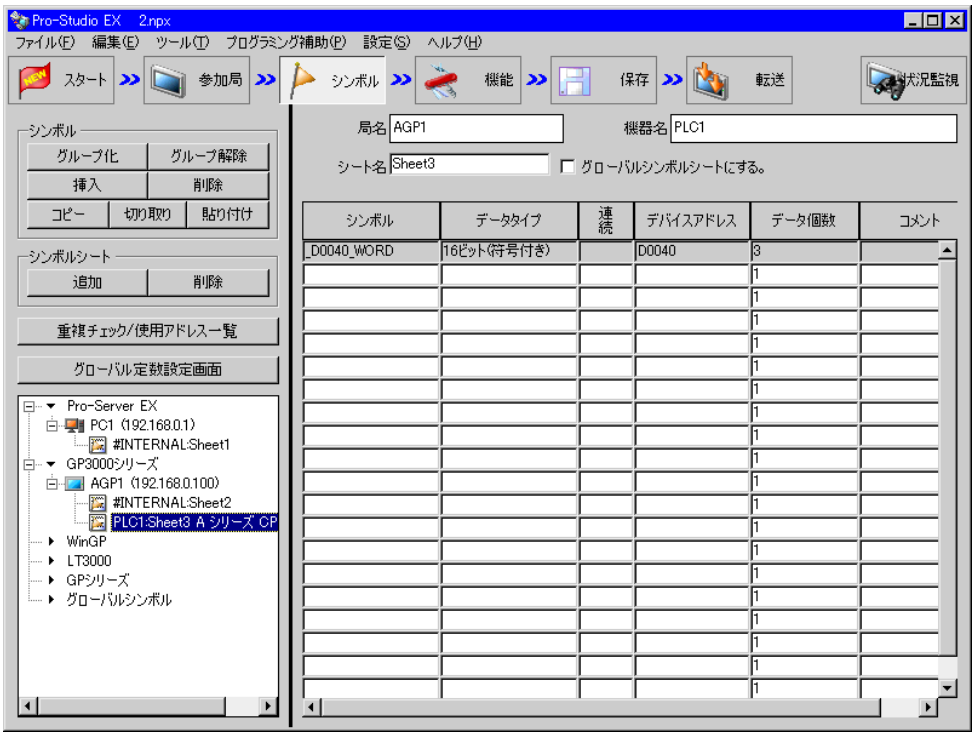
17 ［TextBox］を選択し、［Form1］に貼り付けます。［TextBox］を 3 つ貼り付けます。



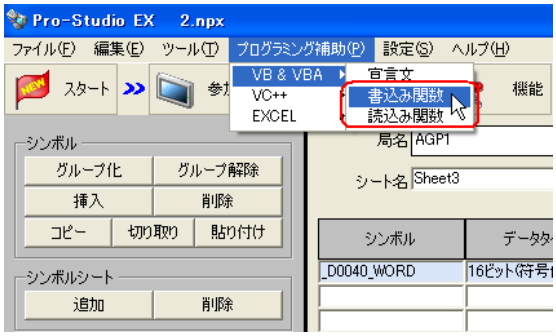
18［CommandButton］を選択し、［Form1］に貼り付けます。



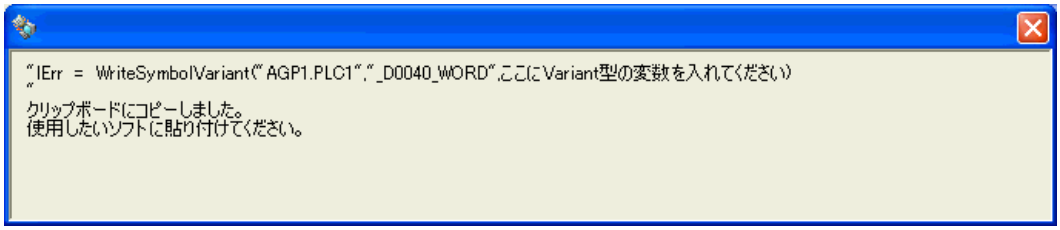
19『Pro-Studio EX』に登録されているシンボルで、書き込み先のシンボル名を選択します。（書き込み先の先頭を選択してください。）



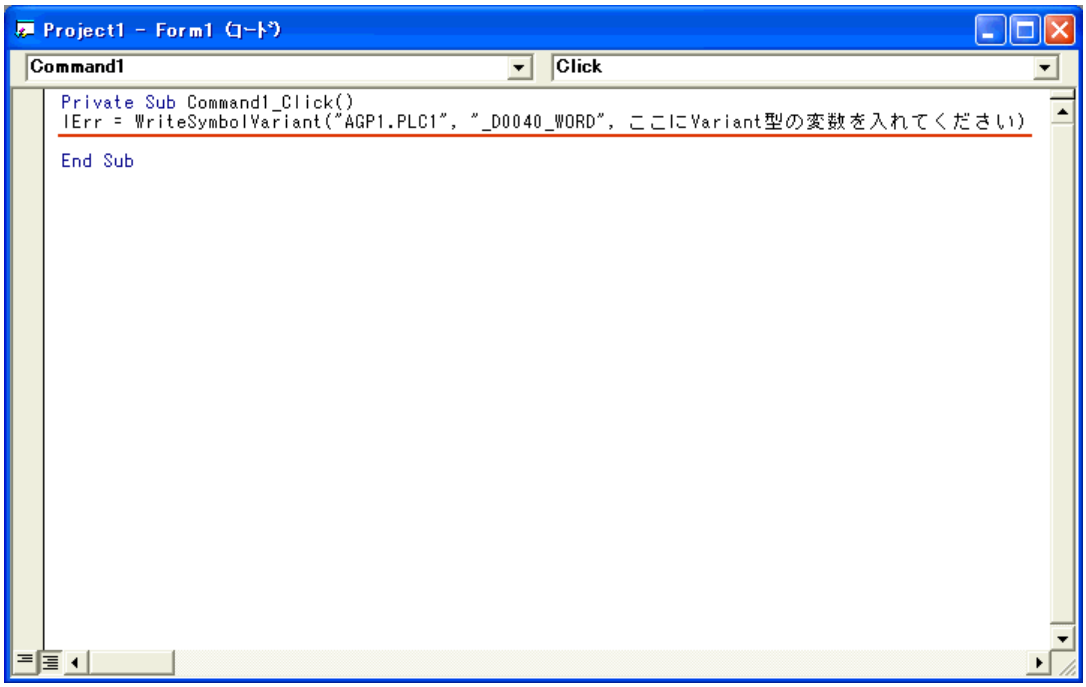
20 メニューの［プログラミング補助］から［VB & VBA］→［書込み関数］を選択します。



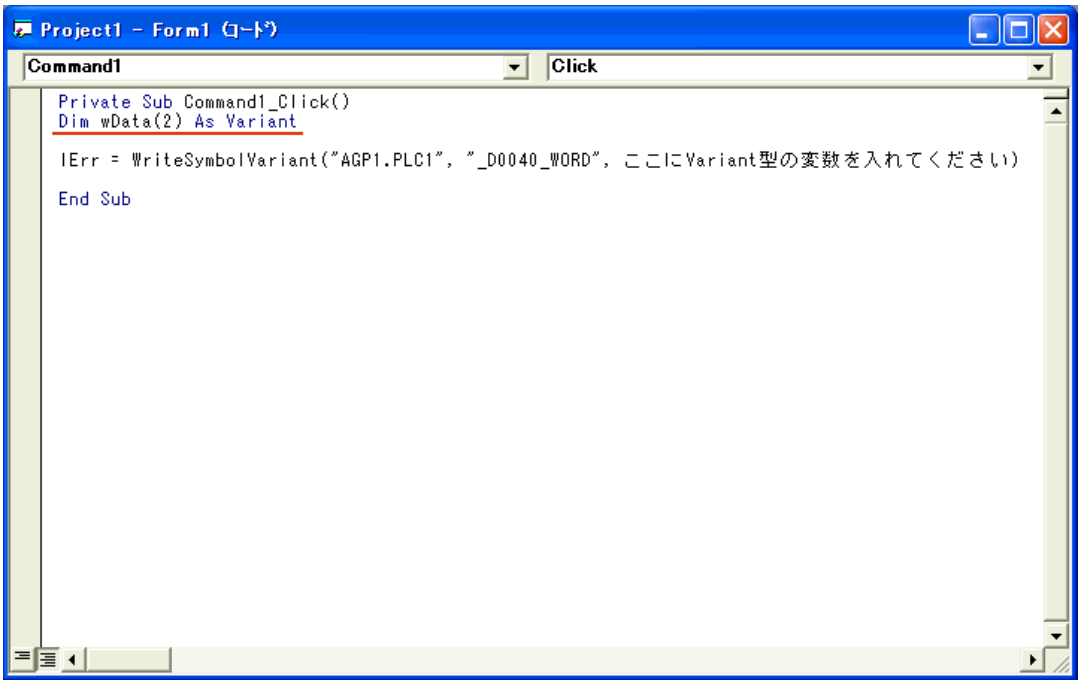
書き込み関数がクリップボードにコピーされます。



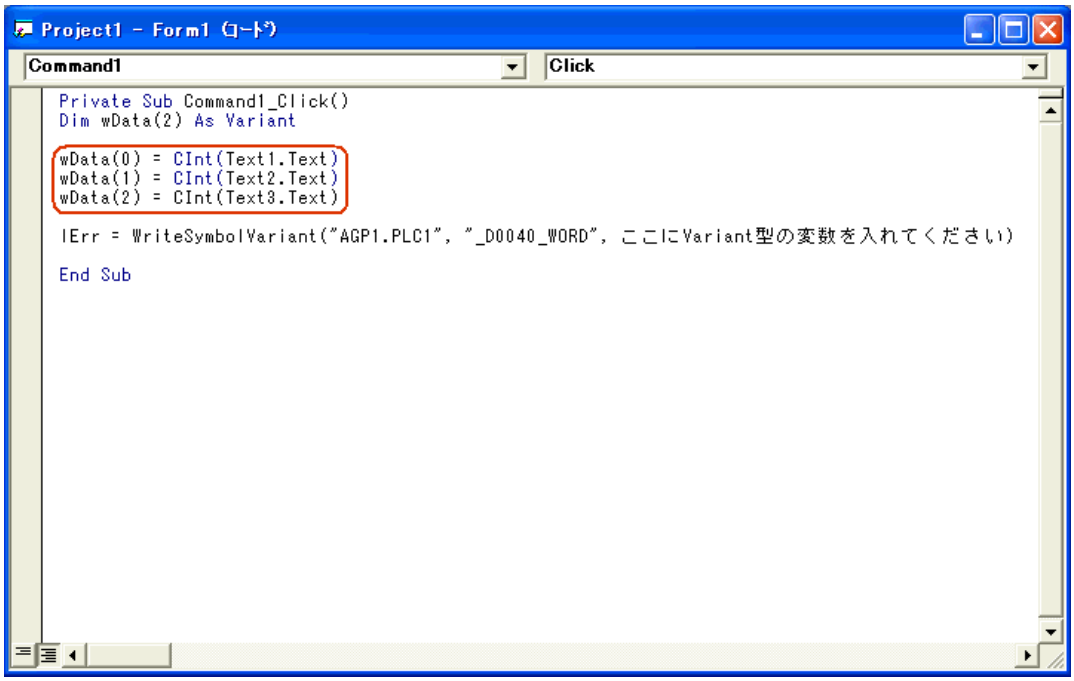
21 ［Form1］上の［Command1］をダブルクリックし、Sub ステートメントと End Sub ステートメントの間にクリップボードの内容（書き込み関数）を貼り付けます。



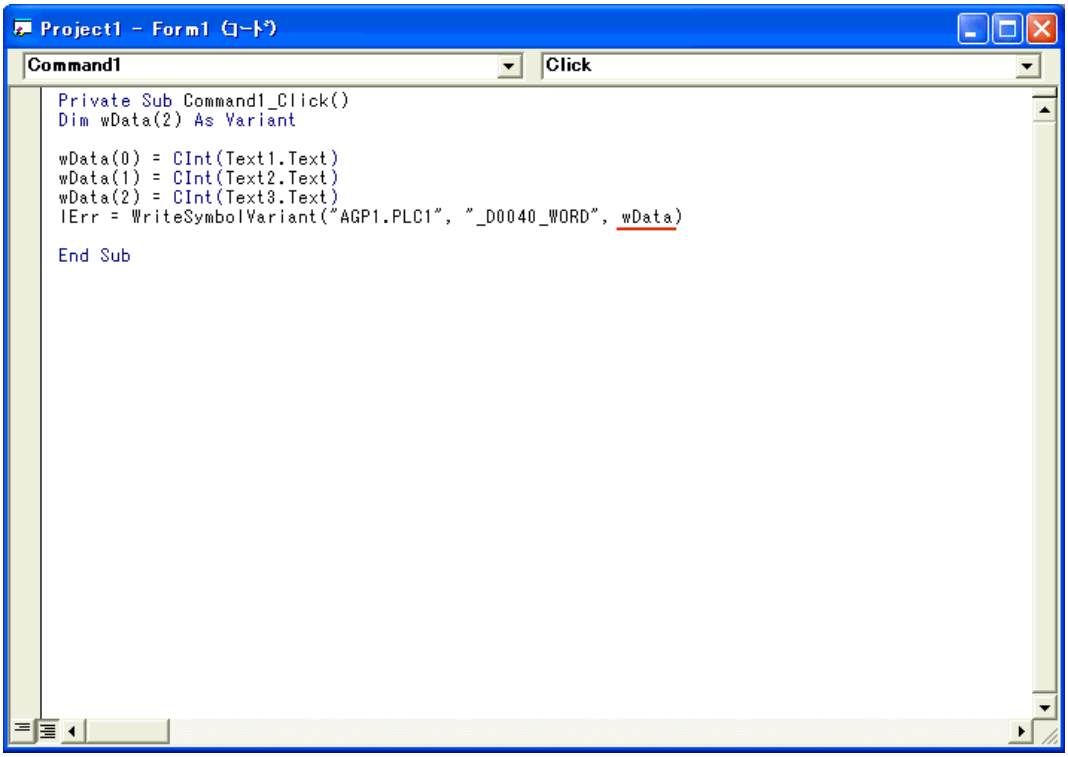
22 書き込むデータを格納するエリア（配列）を宣言します。配列の型（本例では Variant）は、使用するシンボルのデータタイプに合わせてください。



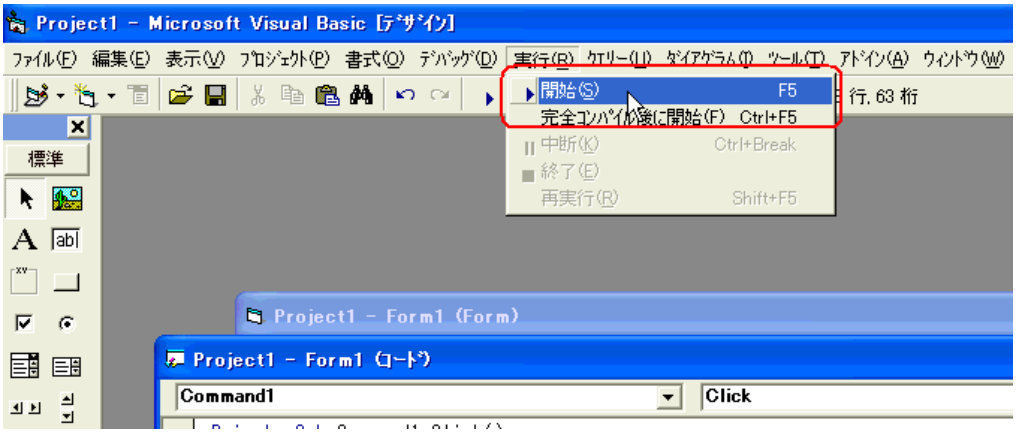
23 [TextBox] に入力されたデータを、配列にセットします。



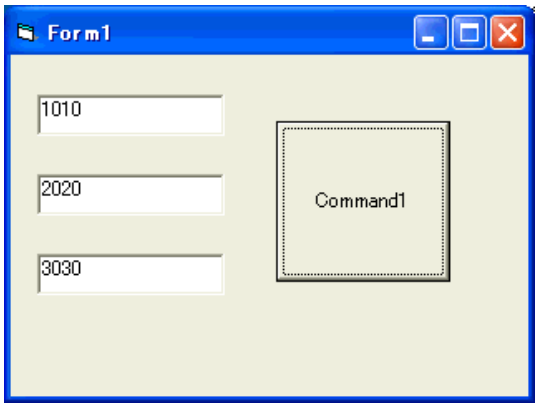
24 書き込みデータがセットされている先頭エリア（wData）を指定します。



25 Microsoft Visual Basic のメニューの［実行］から［開始］を選択します。



26 書き込む値（3 点分）を［TextBox］に入力したあと、［Command1］をクリックすると、シンボル“ _D0040_WORD ”から 3 点分の書き込みが実行されます。

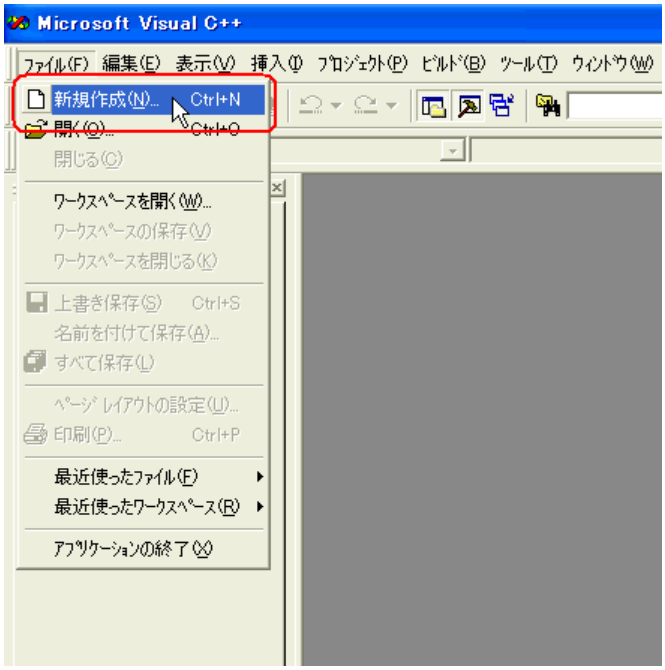


27.10.2 VC 機能補助

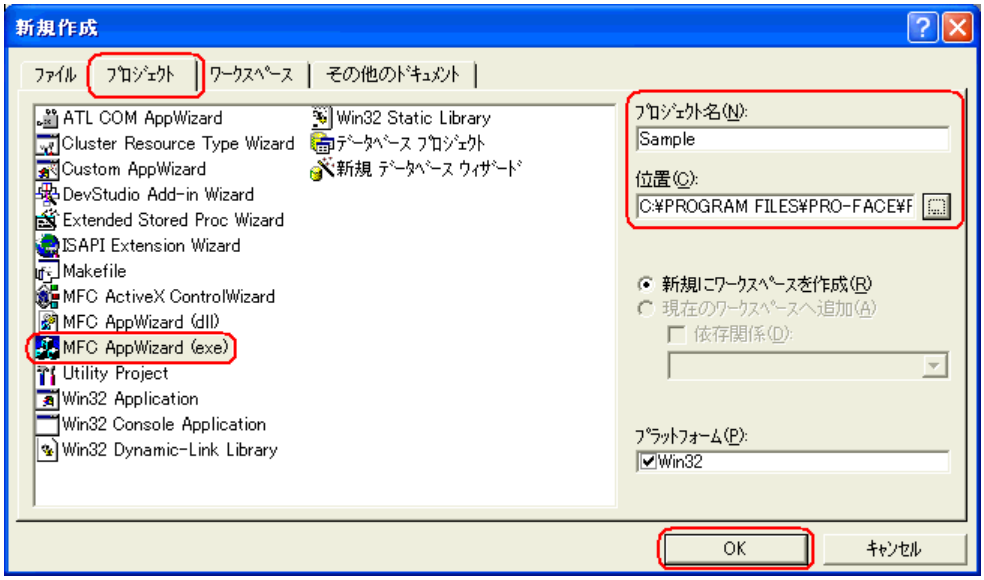
ここでは例として、MFC（Microsoft Foundation Class）を利用したダイアログベースのアプリケーションを作成します。

VC：宣言文

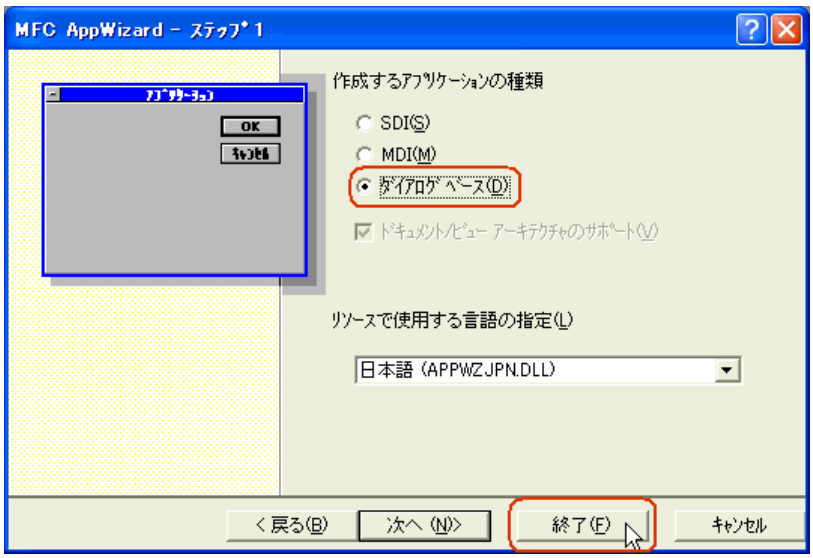
1 Microsoft Visual C++ を起動し、［ファイル］から［新規作成］を選択します。



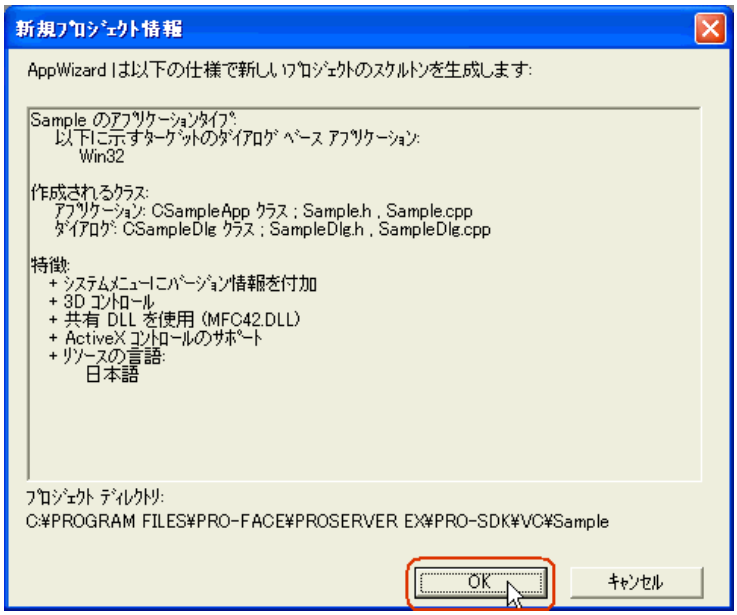
2［プロジェクト］タブで［MFC AppWizard(exe)］を選択したあと、［プロジェクト名］と［位置］を入力し、[OK] ボタンをクリックします。
ここでは、［プロジェクト名］に“ Sample ”を入力し、［位置］には、“ C:¥Program Files¥Pro-face¥Pro-Server EX¥PRO-SDK¥VC ”（Windows Vista の場合は“ C:¥Pro-face¥Pro-Server EX¥PRO-SDK¥VC ”）を入力しています。



3「作成するアプリケーションの種類」の［ダイアログベース］を選択し、［終了］ボタンをクリックします。

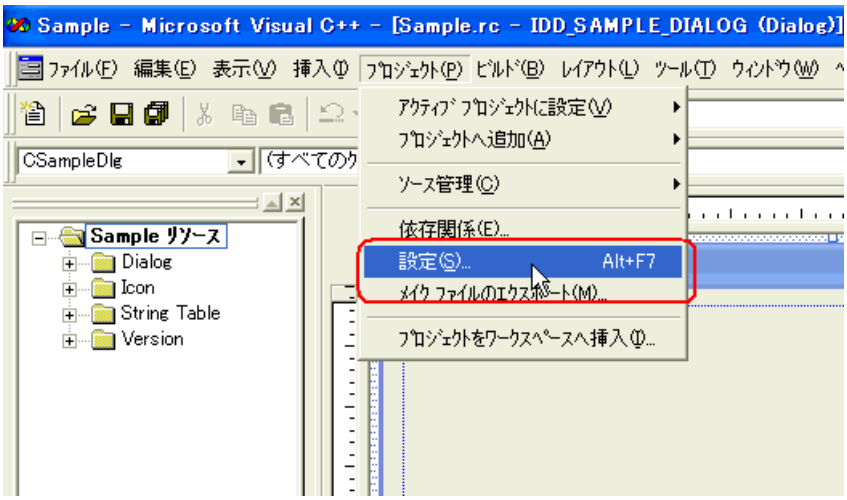


4［OK］ボタンをクリックし、プロジェクトを完成させます。

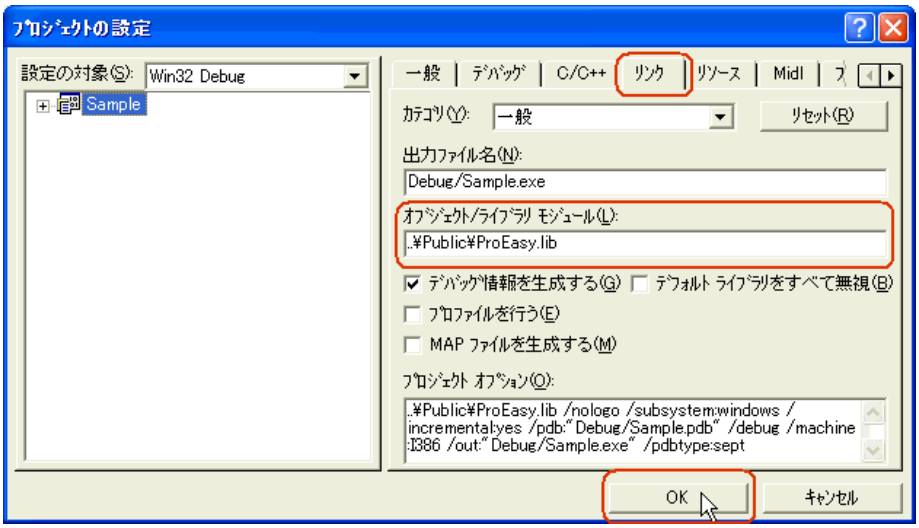


『Pro-Server EX』が提供する読み出し関数／書き込み関数は、DLL として提供されています。その DLL を利用するために、LIB ファイルを指定する必要があります。

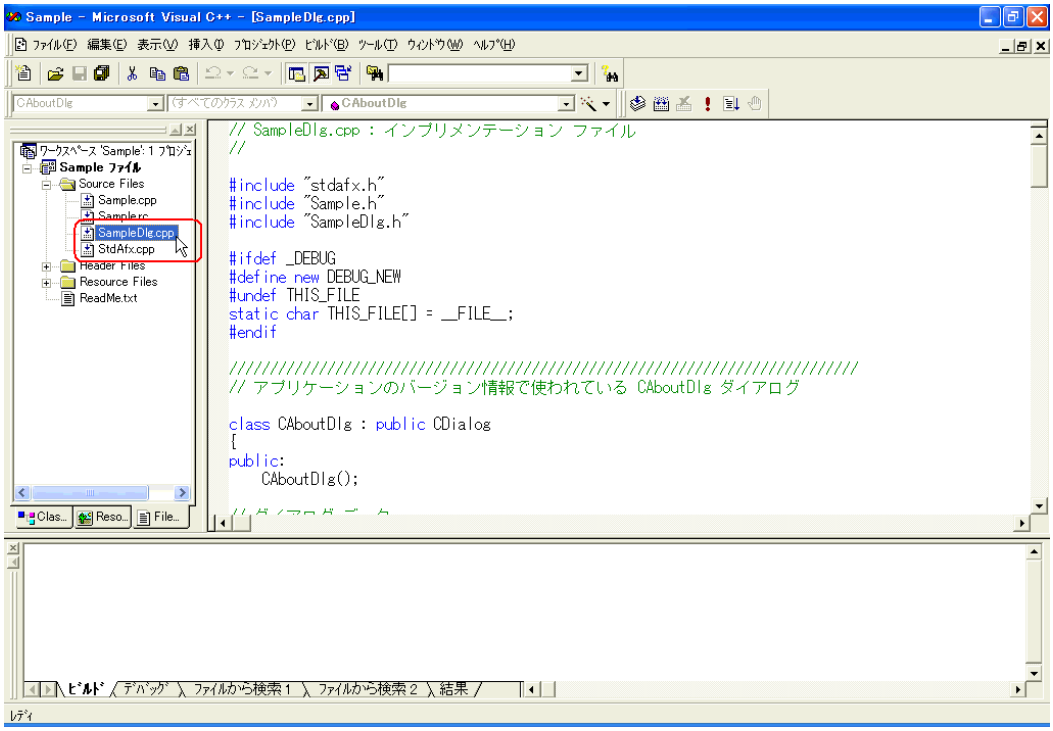
5 Microsoft Visual C++ のメニューの［プロジェクト］から［設定］を選択します。



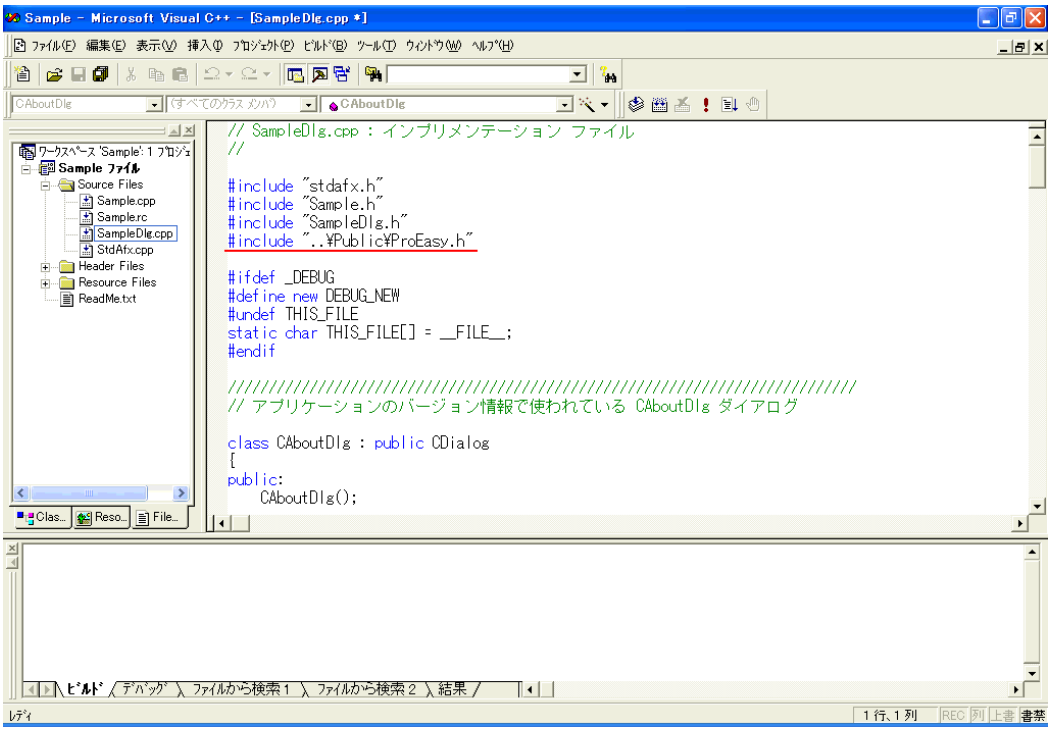
6［リンク］タブの［オブジェクト / ライブラリモジュール］で LIB ファイルを指定します。指定後、［OK］ボタンをクリックします。
LIB ファイル（ProEasy.lib）は、『Pro-Server EX』のインストール先フォルダ内の「PRO-SDK¥Vc¥Public」内に存在しますので、この例では、“ ..¥Public¥ProEasy.lib ”を指定しています。



7『Pro-Server EX』が提供する読み出し関数／書き込み関数を使用するために、ヘッダーファイル（ProEasy.h）をインクルードする必要があります。Microsoft Visual C++ の［ワークスペース］ウィンドウの［FileView］タブをクリックしたあと、SampleDig.cpp ファイルをダブルクリックします。
この例では SampleDig.cpp ファイル内で読み出し／書き込み関数を使用します。



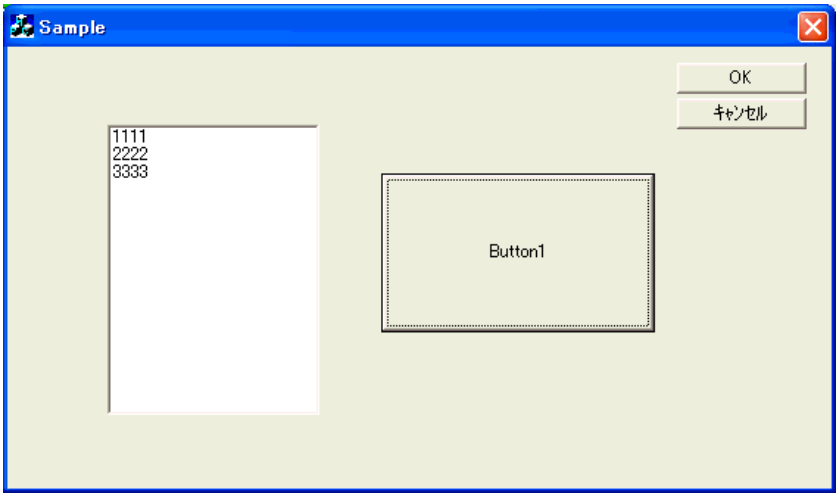
8 SampleDig.cpp ファイルに、#include “ ..¥Public¥ProEasy.h ” を追加すると、関数（読み出し／書き込み関数）の宣言は終了です。



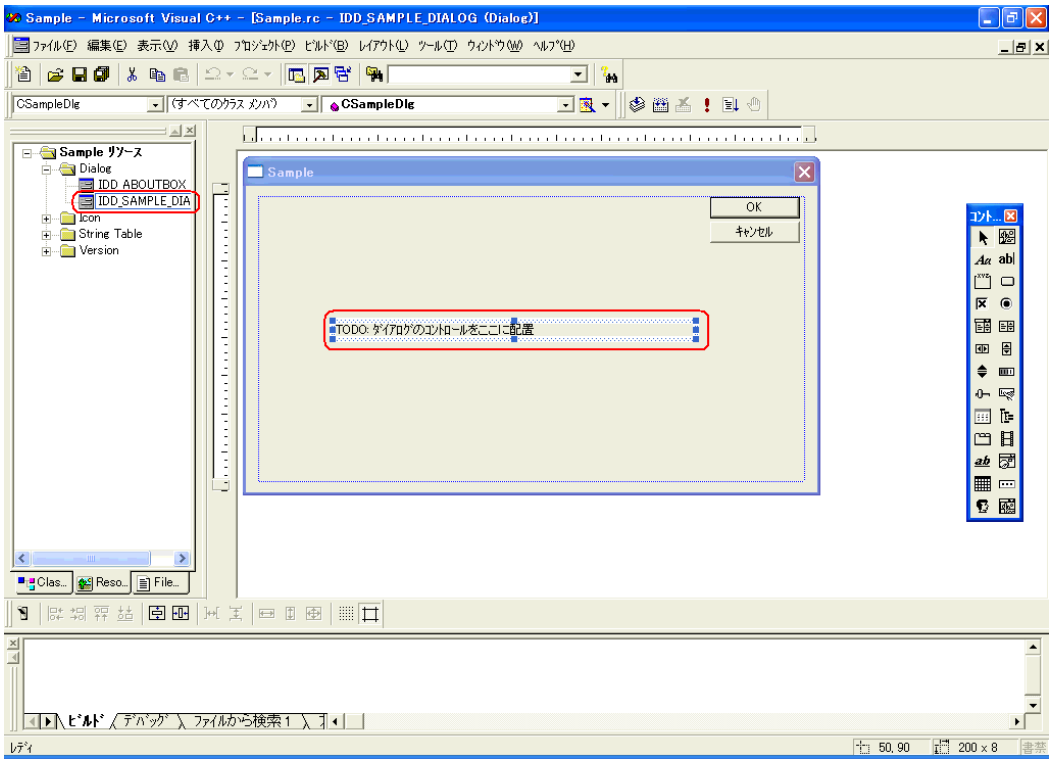
前記 1 ～ 8 の操作は、読み込み／書き込みのいずれの場合でも共通です。
以降の手順については、読み込みの場合と書き込みの場合で手順が異なりますので、個別に説明します。
［読み込み］用アプリケーションの作成については、手順 9 ～ 30 をご覧ください。
［書き込み］用アプリケーションの作成については、手順 31 ～ 47 をご覧ください。

〔読み込み〕用アプリケーションの作成

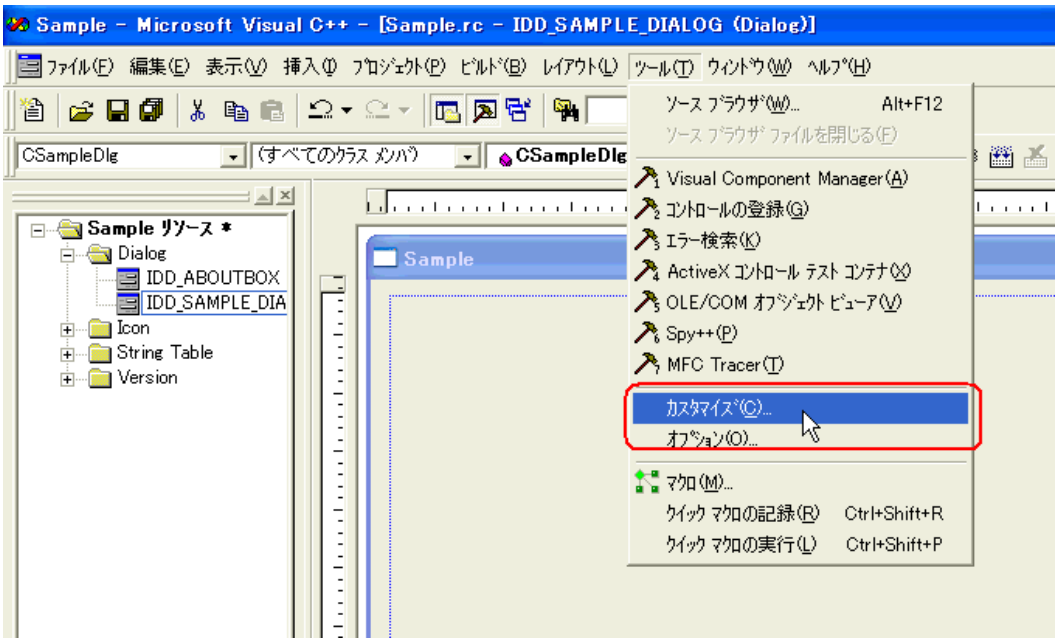
ここでは、［Button1］をクリックすると、3 点分のデータ（16 ビット符号付）を読み出して表示するアプリケーションについて説明します。



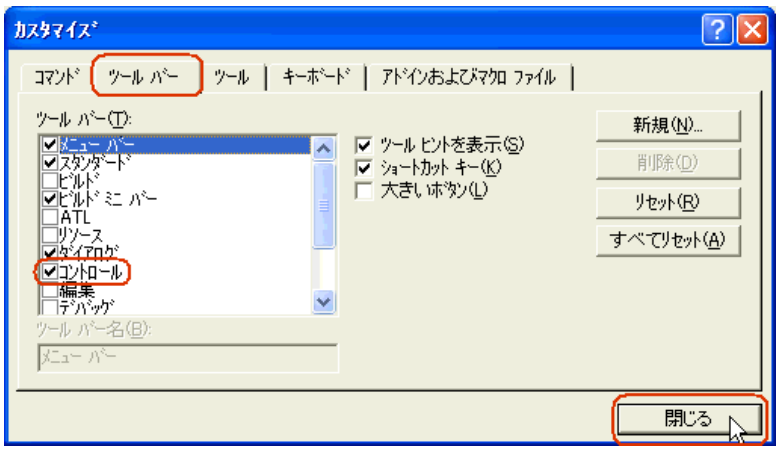
9 Microsoft Visual C++ の［ワークスペース］ウィンドウの［ResourceView］タブをクリックしたあと、［IDD_SAMPLE_DIALOG］をダブルクリックします。
ダイアログ中央の［スタティックテキスト］を選択し、削除します。



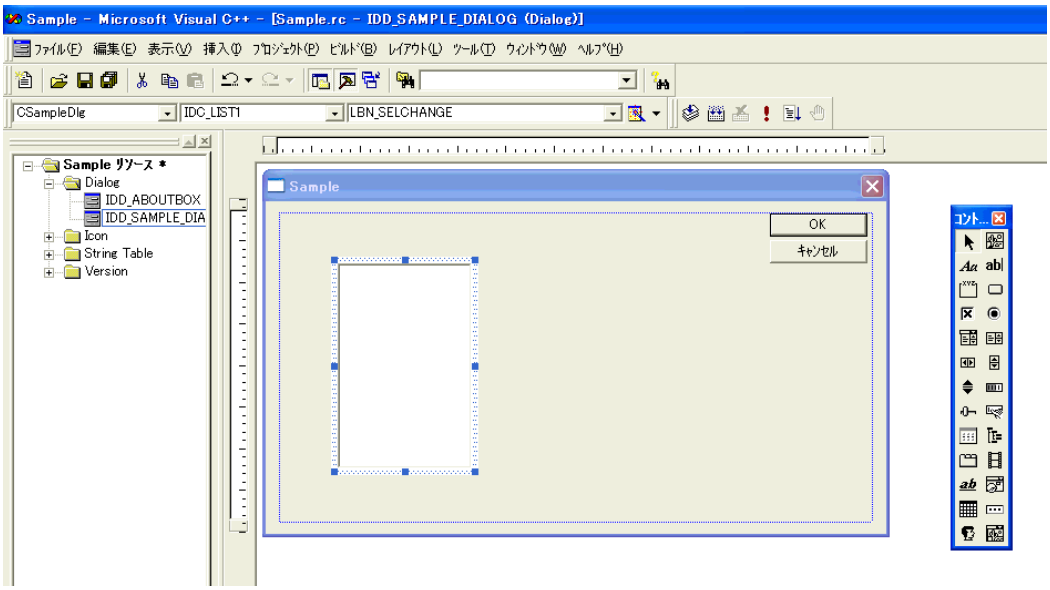
10 Microsoft Visual C++ のメニューの［ツール］から［カスタマイズ］を選択します。



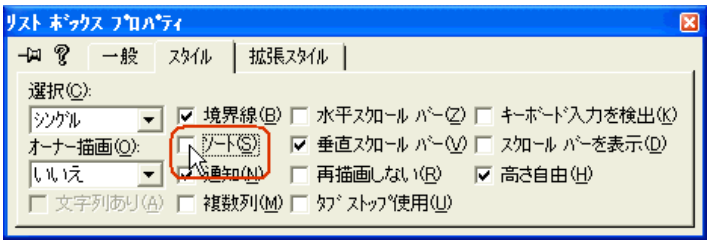
11［ツールバー］タブで［コントロール］にチェックを入れ、［閉じる］ボタンをクリックします。



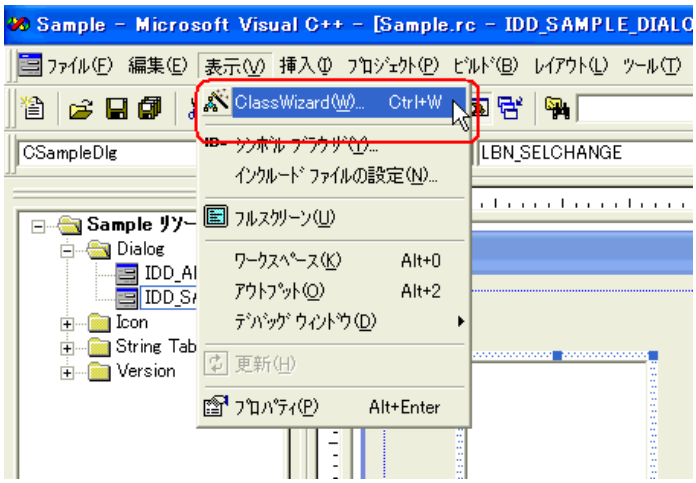
12［リストボックス］を選択し、ダイアログに貼り付けます。



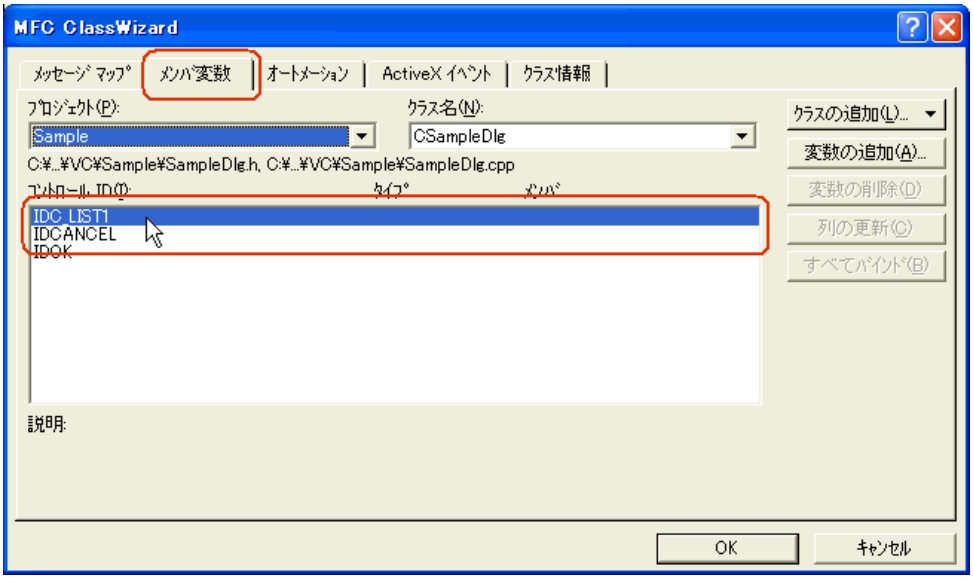
13 貼り付けた［リストボックス］を右クリックし、［プロパティ］を選択します。［リストボックスプロパティ］ダイアログが表示されますので、［ソート］のチェックを外します。



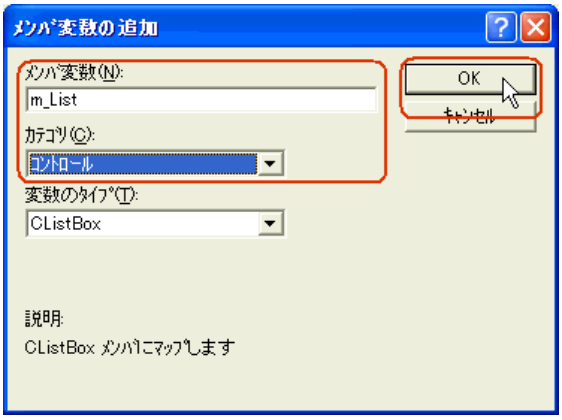
14 Microsoft Visual C++ のメニューの［表示］から［ClassWizard］を選択します。



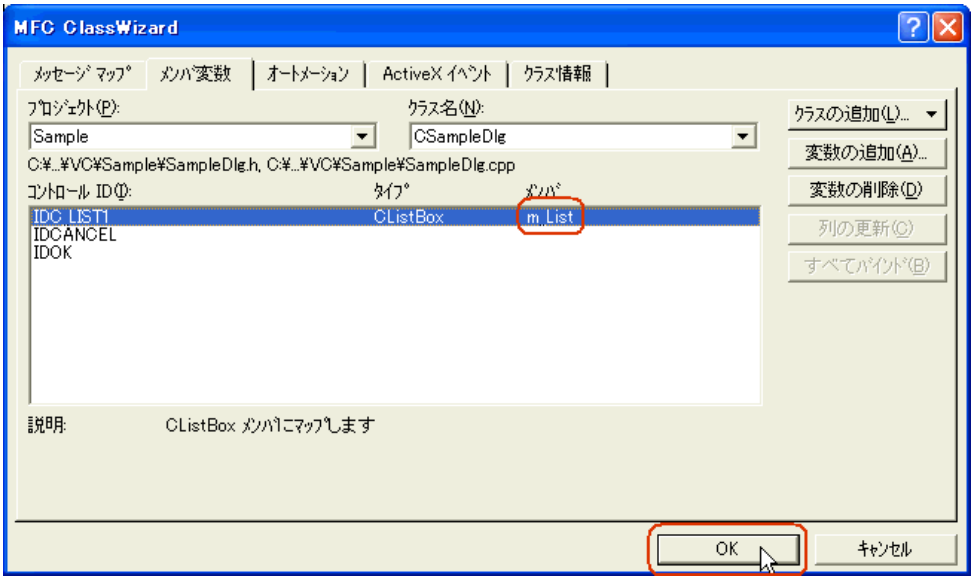
15［メンバ変数］タブを選択し、［コントロール ID］の“ IDC_LIST1 ”を選択します。



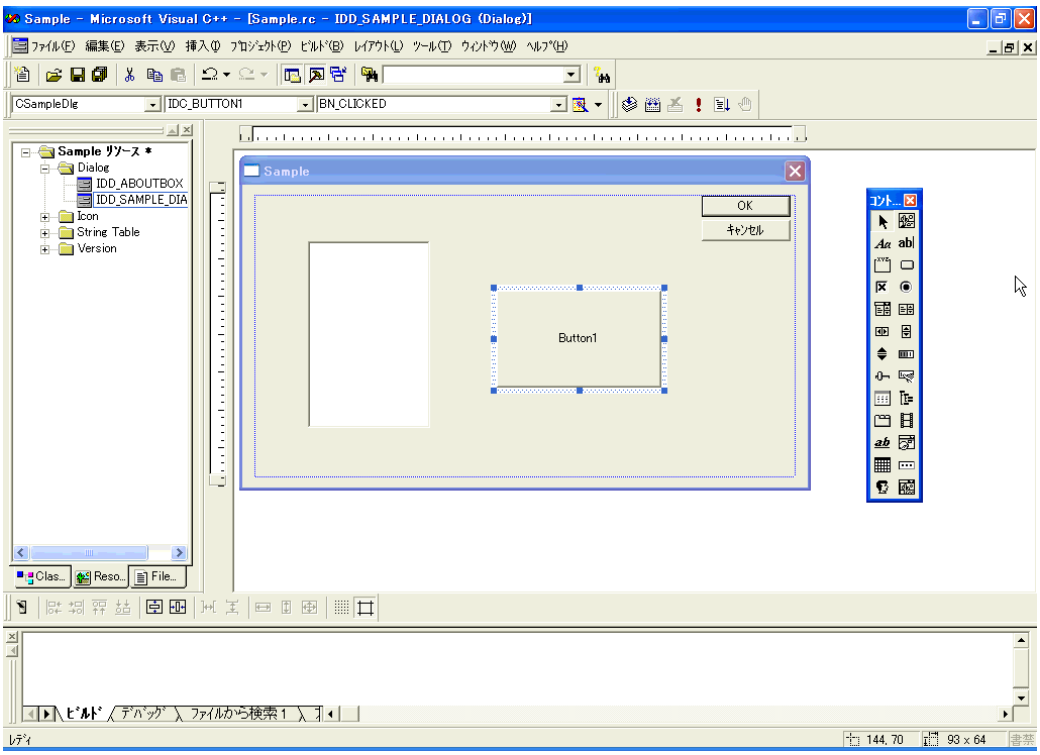
16［変数の追加］をクリックし、［メンバ変数］に“ m_List ”を入力し、［カテゴリ］に“ コントロール ”を選択したあと、［OK］ボタンをクリックします。



17 メンバ変数が追加されていることを確認したあと、［OK］ボタンをクリックします。

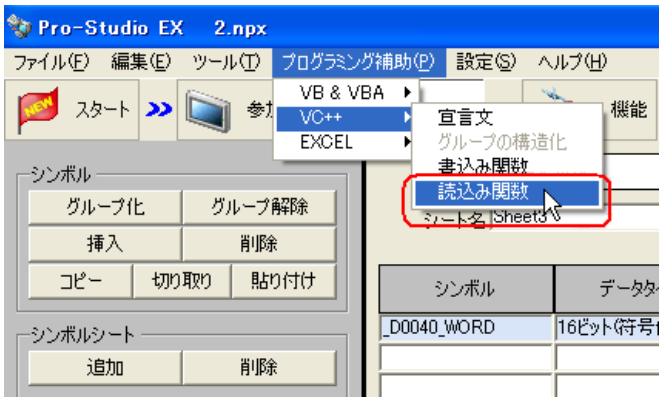


18［ボタン］を選択し、ダイアログに貼り付けます。

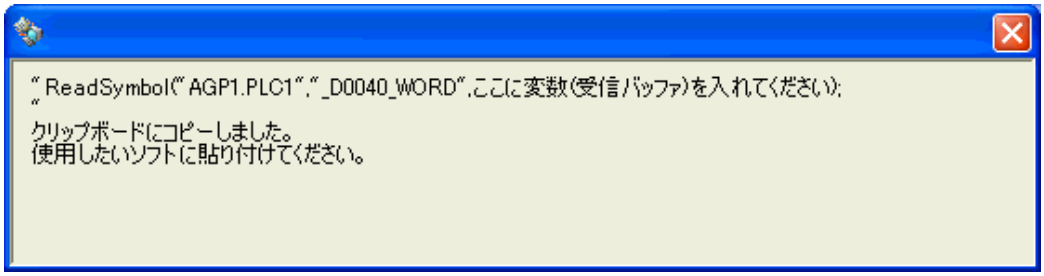


19 『Pro-Studio EX』に登録されているシンボルで、読み込みたいシンボル名を選択します。（読み込み先の先頭を選択してください。）

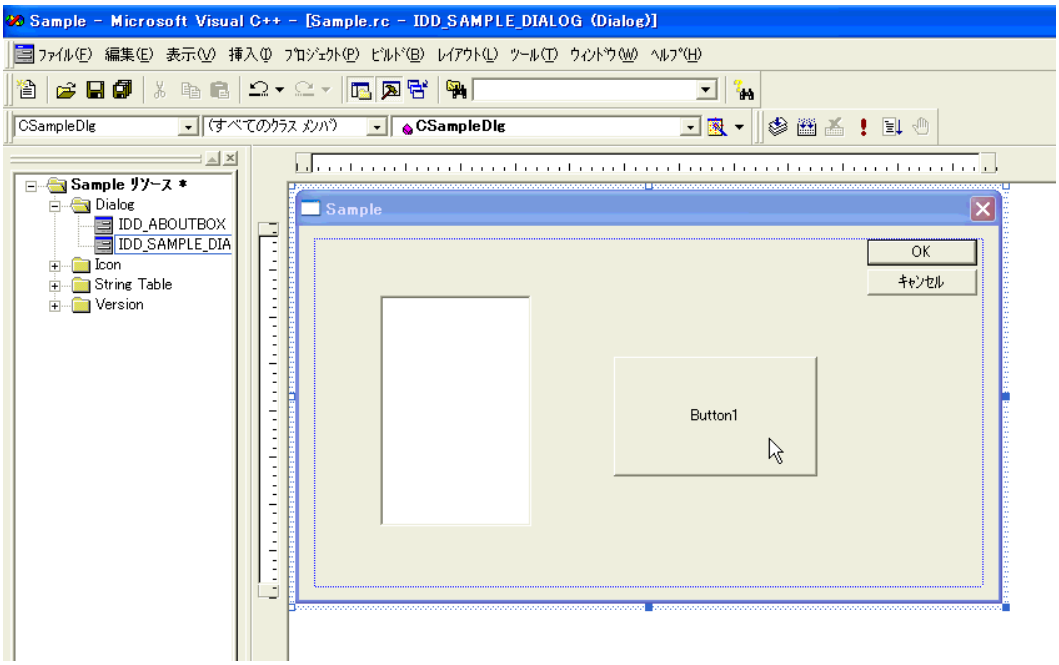
20 メニューの『プログラミング補助』から、［VC++］→［読込み関数］を選択します。



読み込み関数がクリップボードにコピーされます。

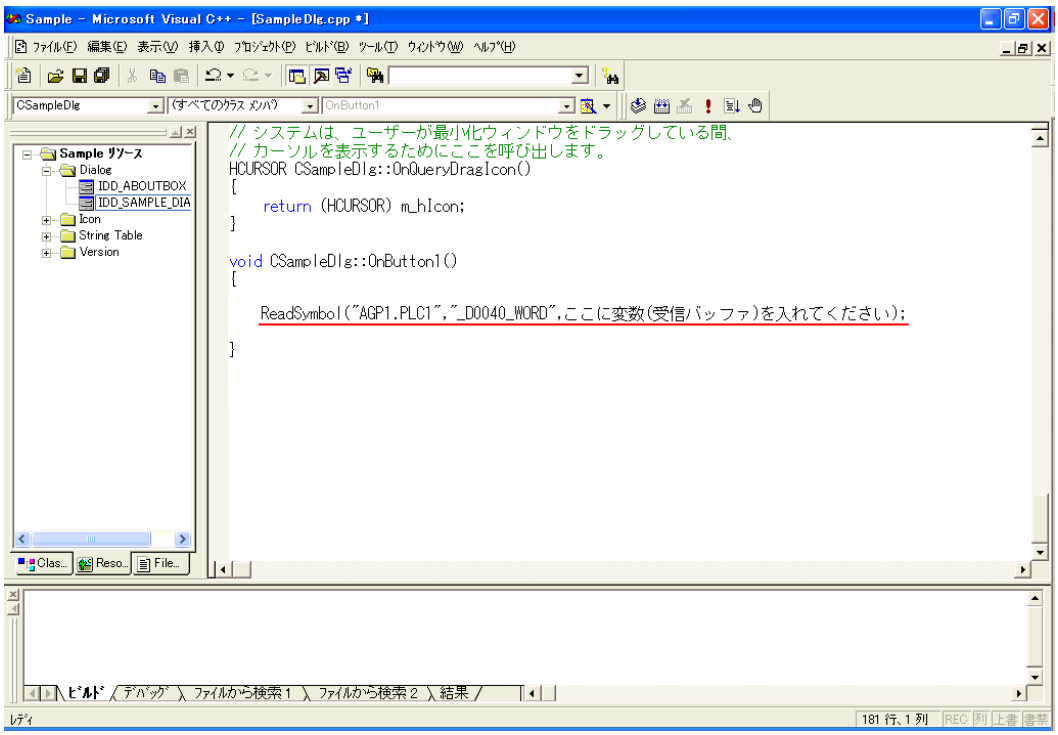


21 Microsoft Visual C++ の［ダイアログ］に貼り付けた［Button1］をダブルクリックします。

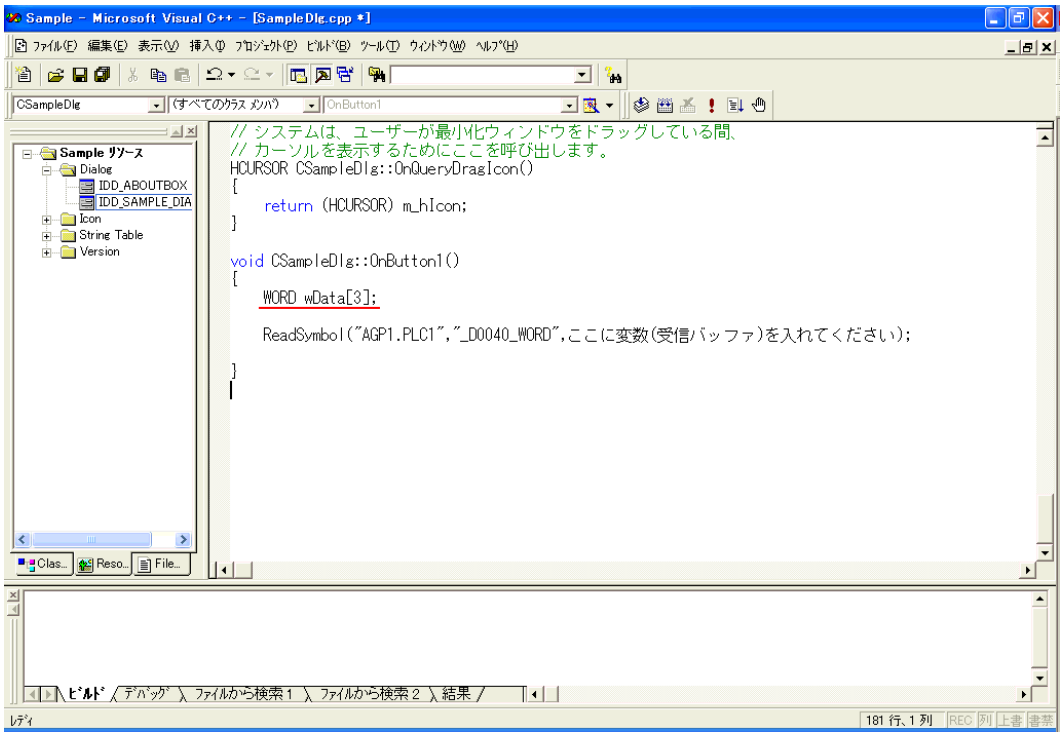


22［OK］ボタンをクリックします。

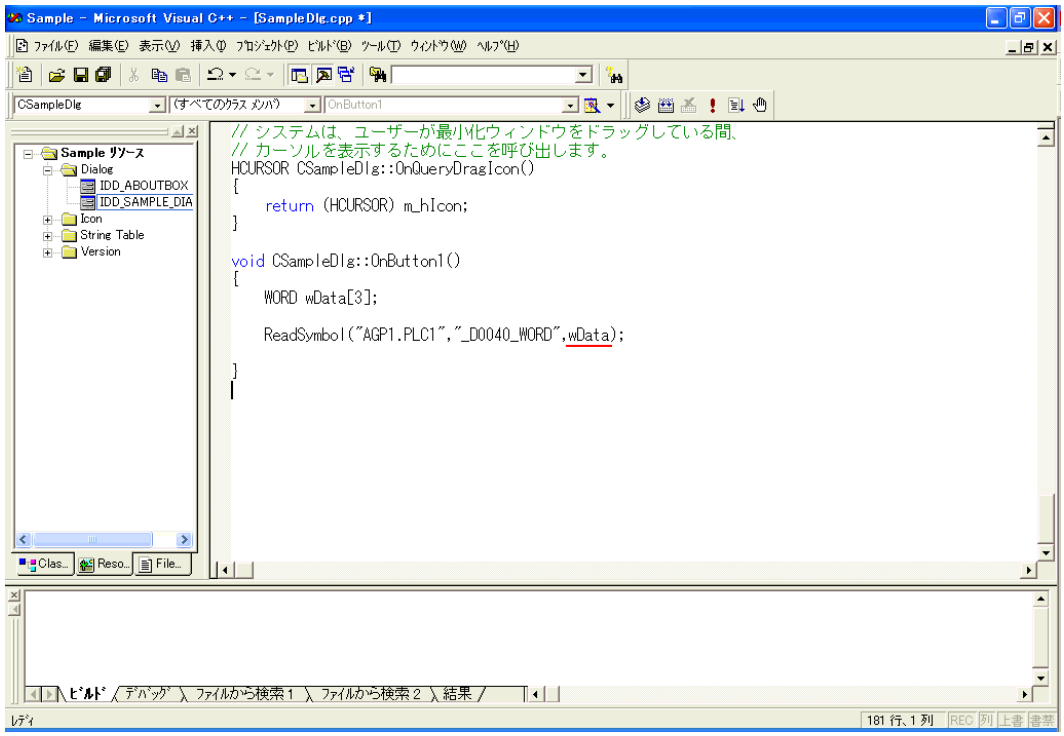
23 OnButton1 メンバ関数内にクリップボードの内容（読み込み関数）を貼り付けます。



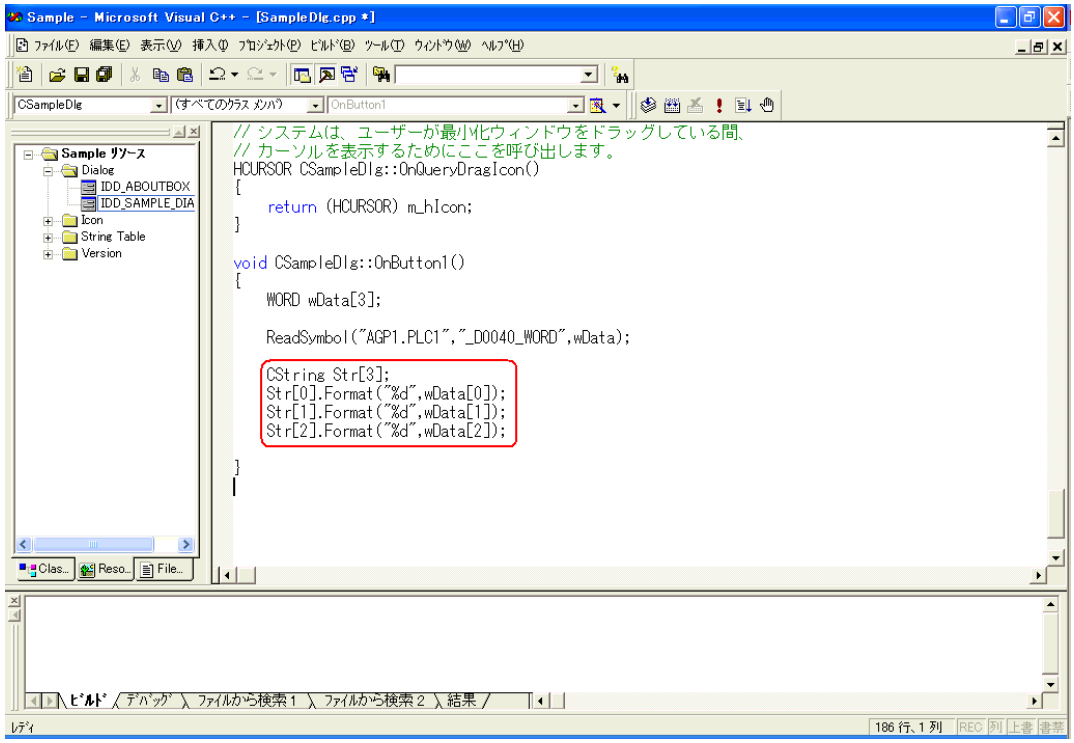
24 読み込んだデータを格納するエリア（配列）を宣言します。



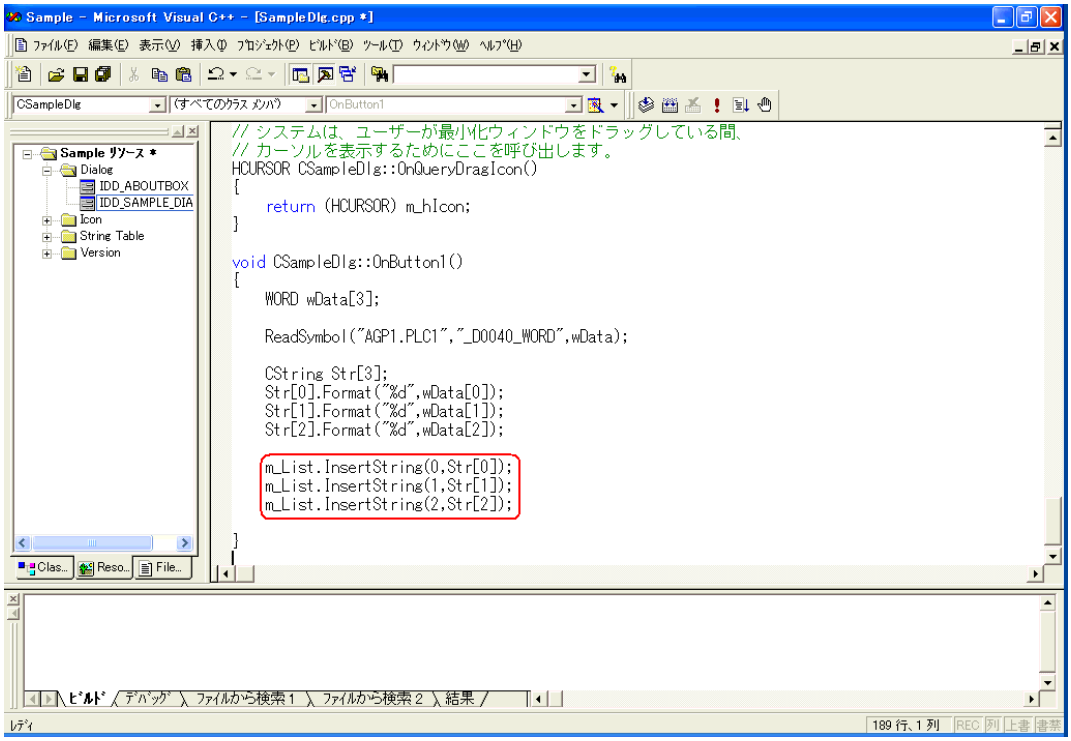
25 読み込んだデータを格納する先頭エリア（wData）を指定します。



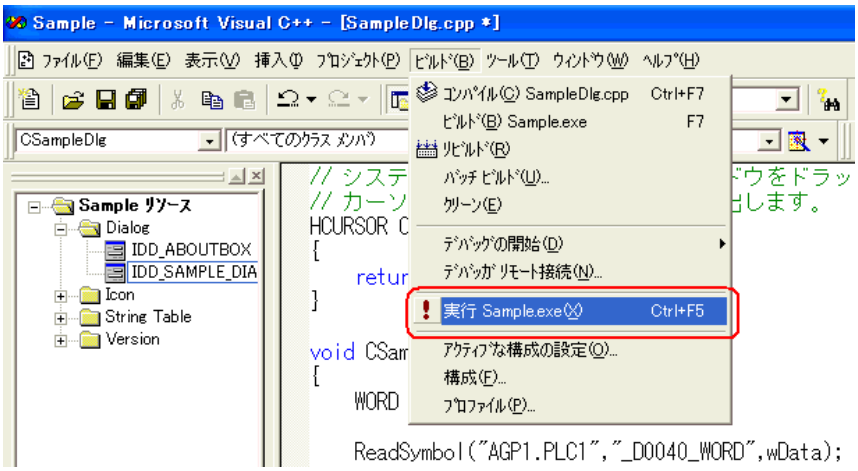
26 読み込んだデータ 3 点分（wData(0)、wData(1)、wData(2)）を、リストボックスに表示するために、一旦 CString 型の文字列に変換します。



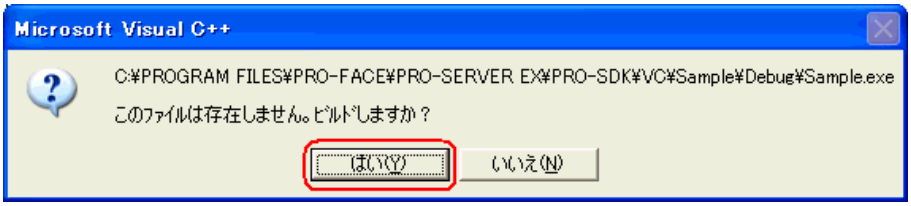
27 読み込んだデータ（文字列に変換されています）を、リストボックス（m_List）に順次表示します。



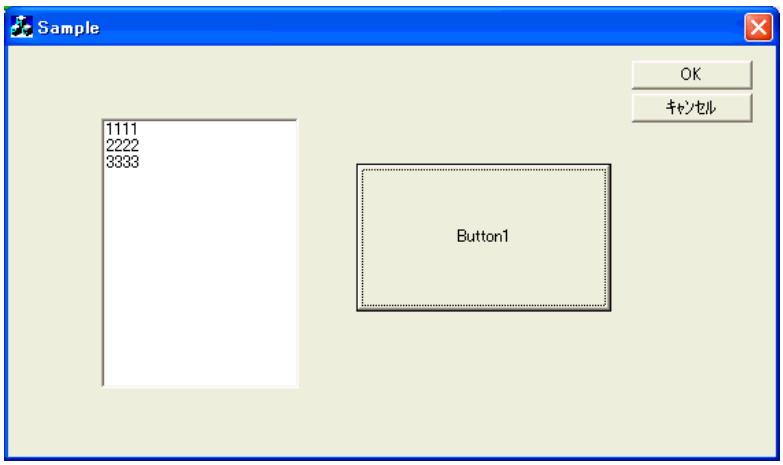
28 Microsoft Visual C++ のメニューの［ビルド］から［実行 Sample.exe］を選択します。



29［はい］ボタンをクリックします。



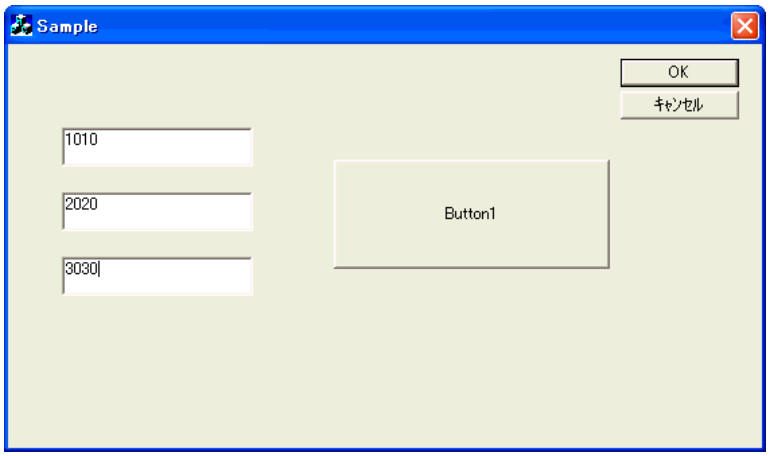
30［Button1］をクリックすると、シンボル“ _D0040_WORD ”から 3 点分のデータがリストボックスに表示されます。



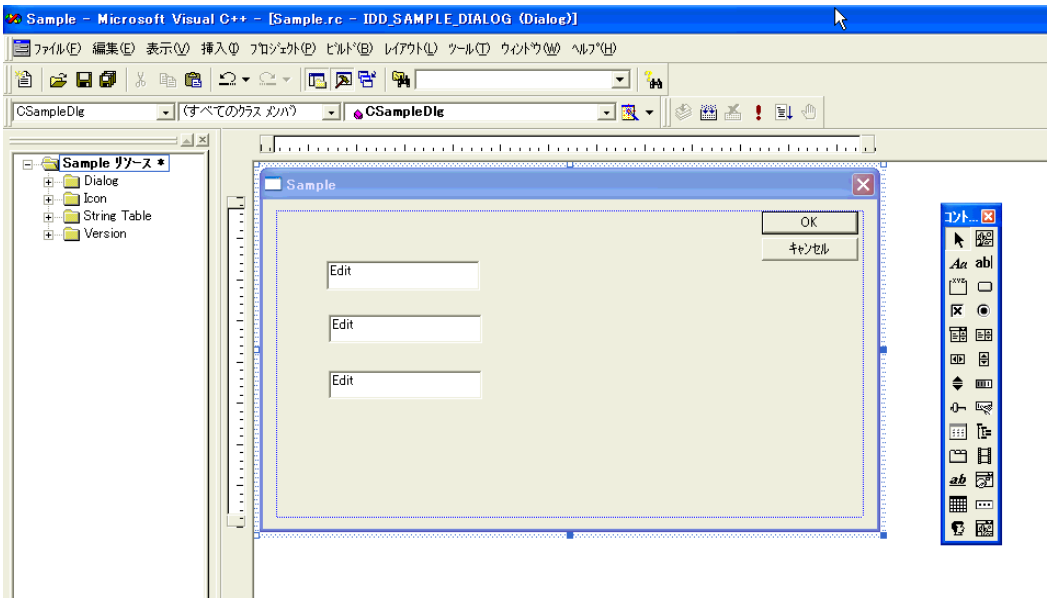
〔書き込み〕用アプリケーションの作成

ここでは、［Button1］をクリックすると、入力された 3 点分のデータを書き込むアプリケーションについて説明します。

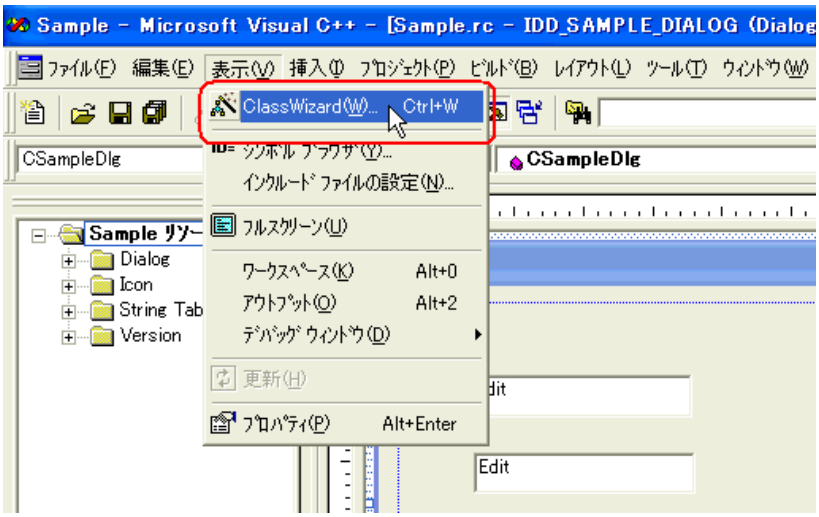
手順の 9 ～ 11 までは、読み込み時と同じ操作を行います。



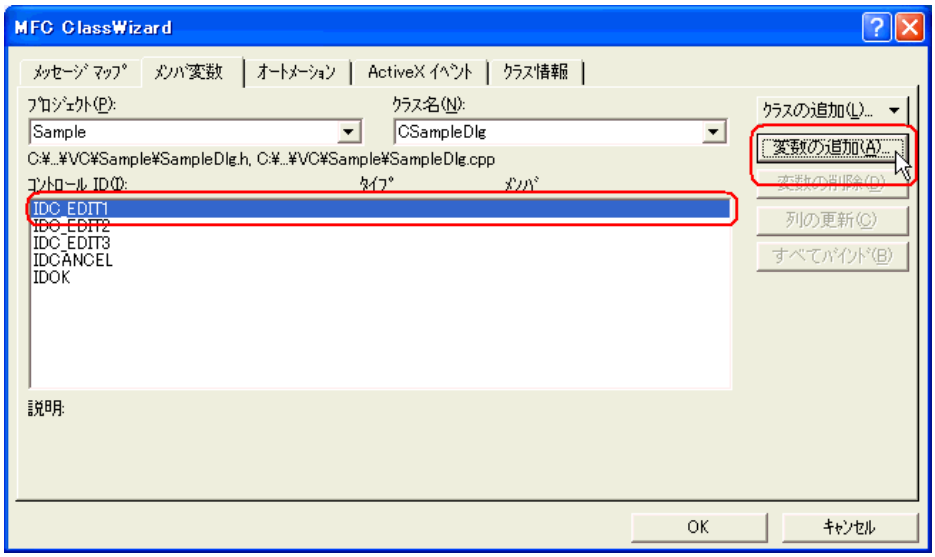
31 ［エディットボックス］を選択し、［ダイアログ］に貼り付けます。［エディットボックス］は 3 つ貼り付けます。



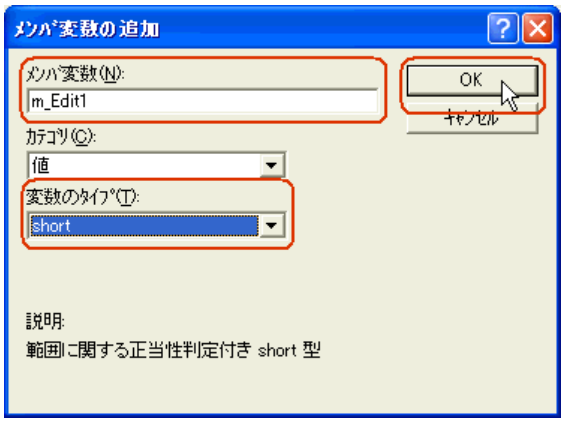
32 Microsoft Visual C++ のメニューの［表示］から［ClassWizard］を選択します。



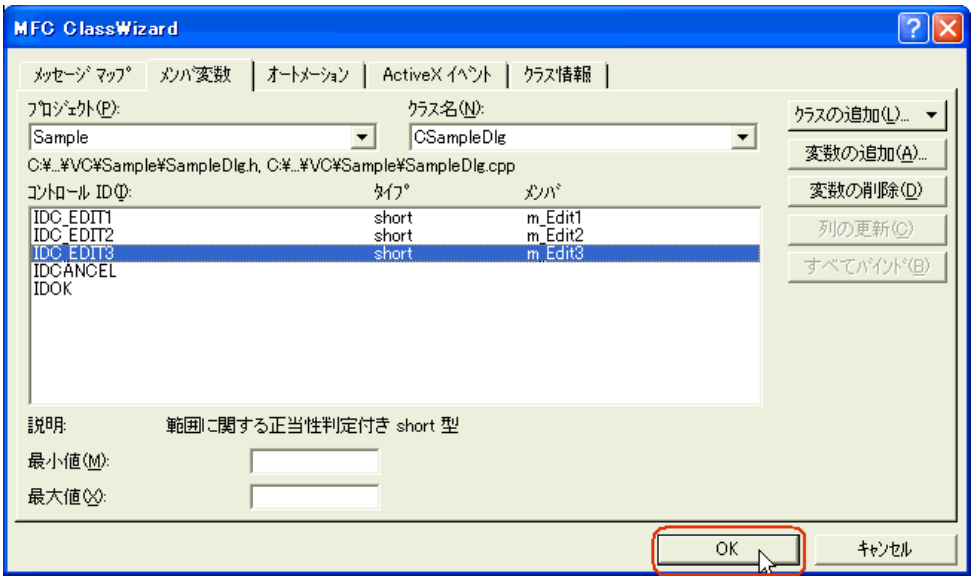
33［メンバ変数］タブで［コントロール ID］の“ IDC_EDIT1 ”を選択し、［変数の追加］ボタンをクリックします。



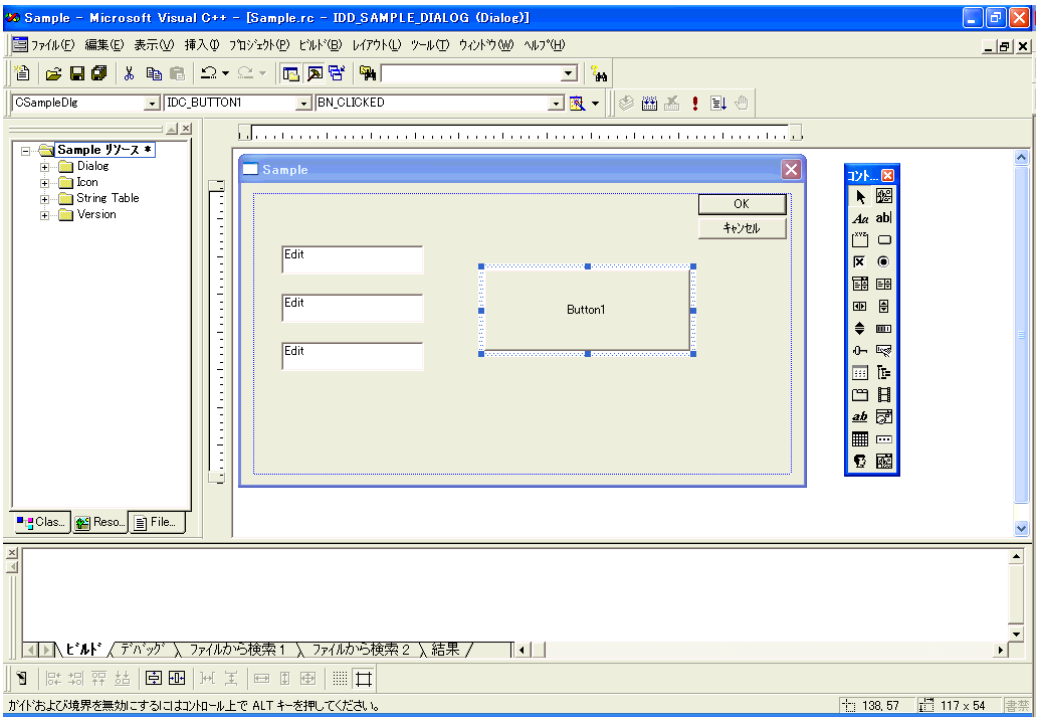
34［メンバ変数］に“ m_Edit1 ”を入力し、［変数のタイプ］に“ short ”を選択したあと、［OK］ボタンをクリックします。
残り 2 つの［エディットボックス］についても、33 ～ 34 の操作を繰り返します。ただし、メンバ変数名はそれぞれ“ m_Edit2 ”、“ m_Edit3 ”を指定してください。



35［OK］ボタンをクリックします。

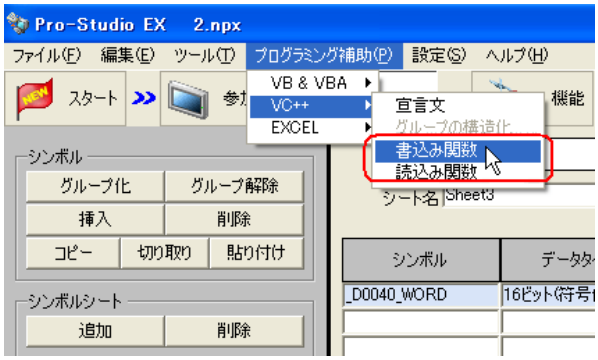


36［ボタン］を選択し、［ダイアログ］に貼り付けます。

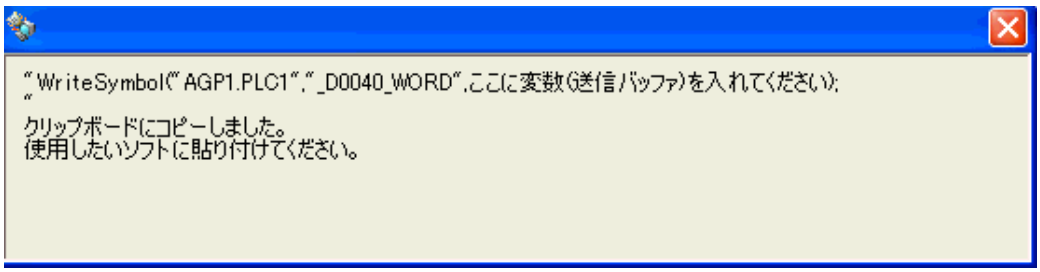


37『Pro-Studio EX』に登録されているシンボルで、書き込み先のシンボル名を選択します。（書き込み先の先頭を選択してください。）

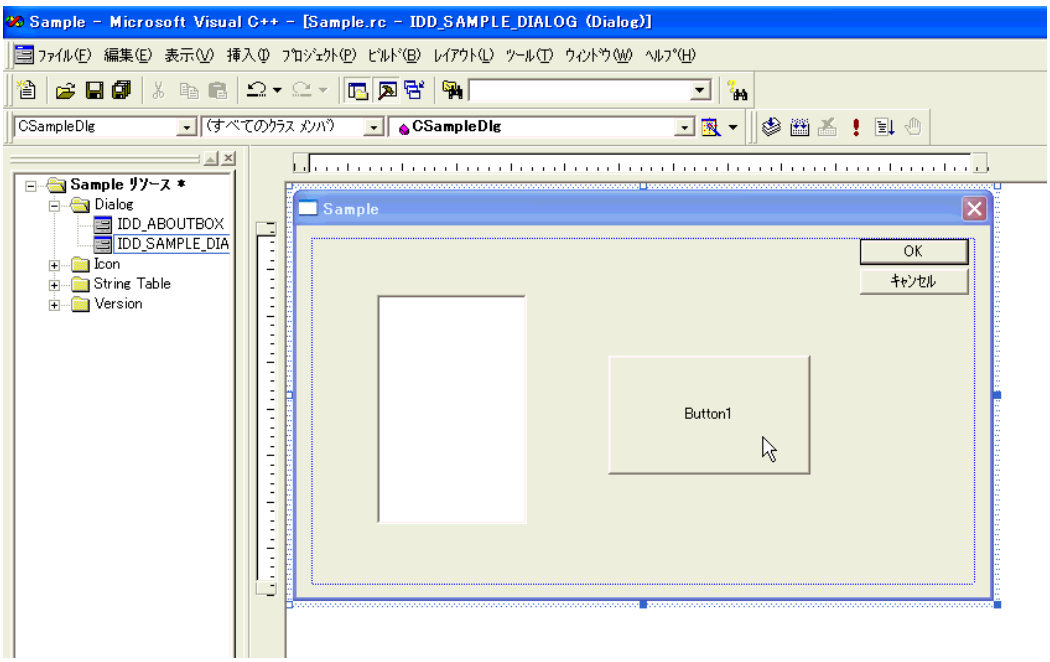
38 メニューの［プログラミング補助］から、［VC++］→［書込み関数］を選択します。



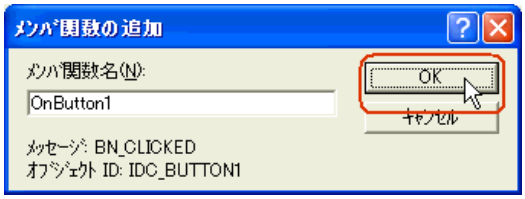
書き込み関数がクリップボードにコピーされます。



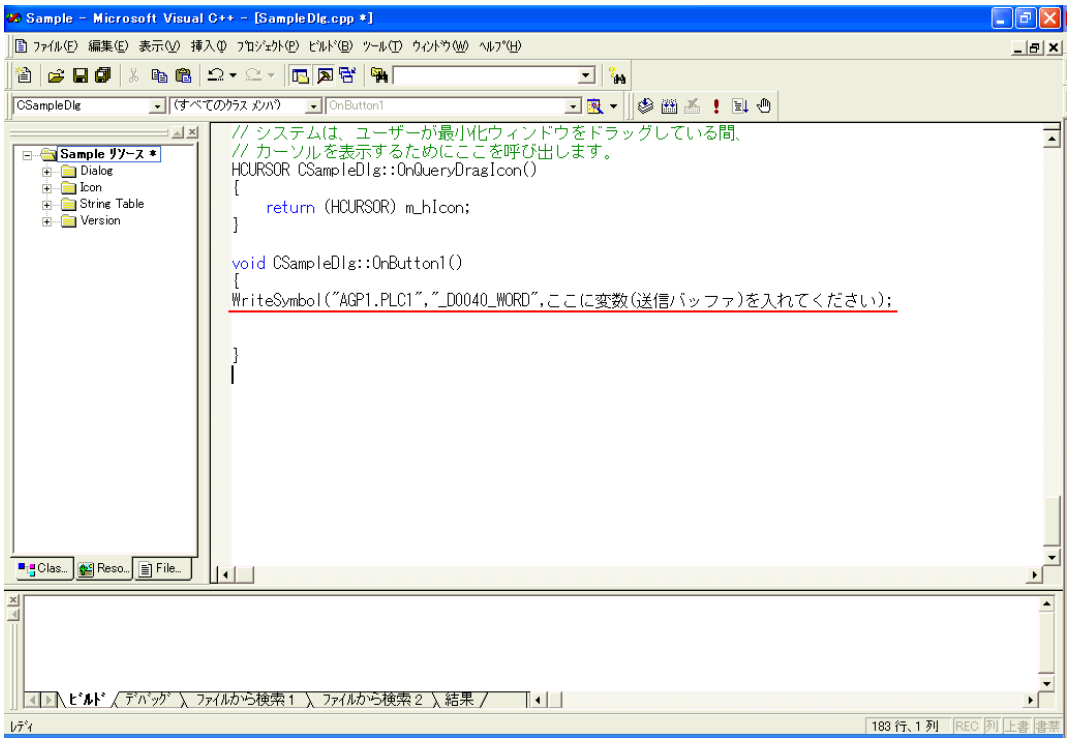
39 Microsoft Visual C++ の［ダイアログ］に貼り付けた［Button1］をダブルクリックします。



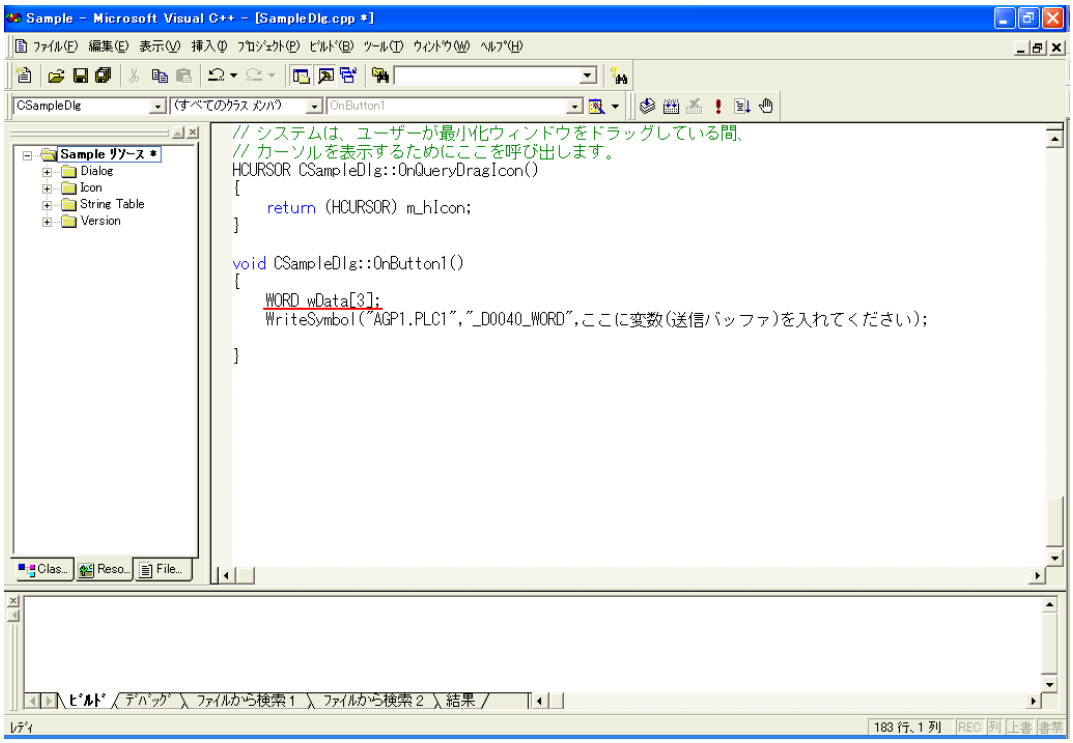
40［OK］ボタンをクリックします。



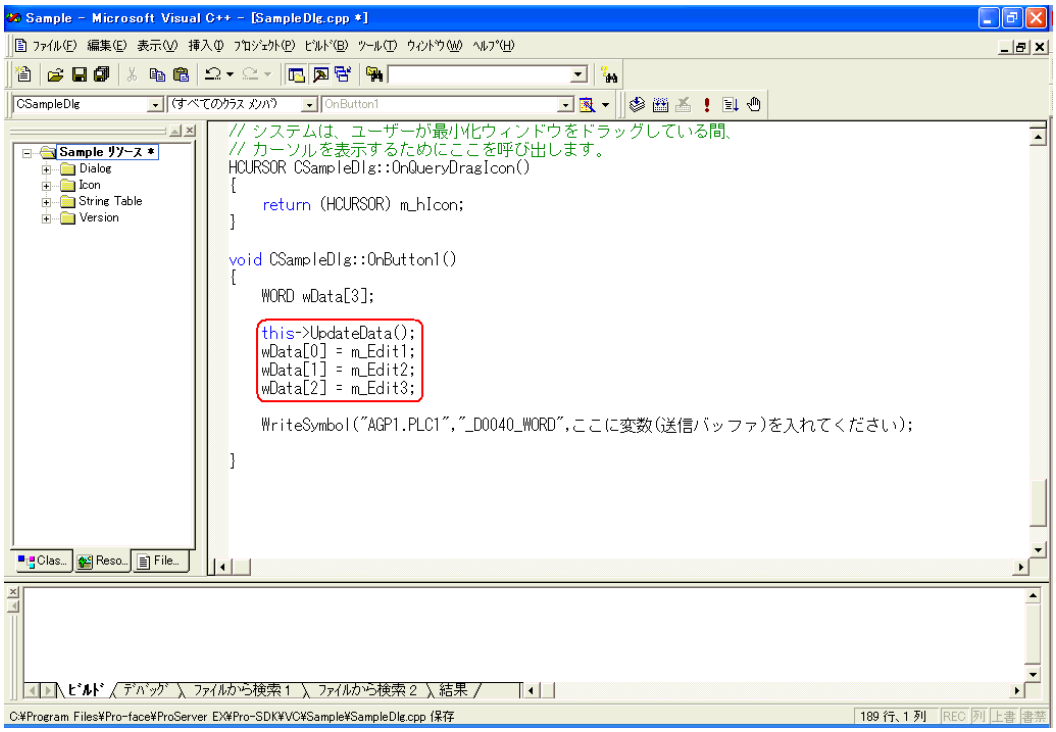
41 OnButton1 メンバ関数内にクリップボードの内容（書き込み関数）を貼り付けます。



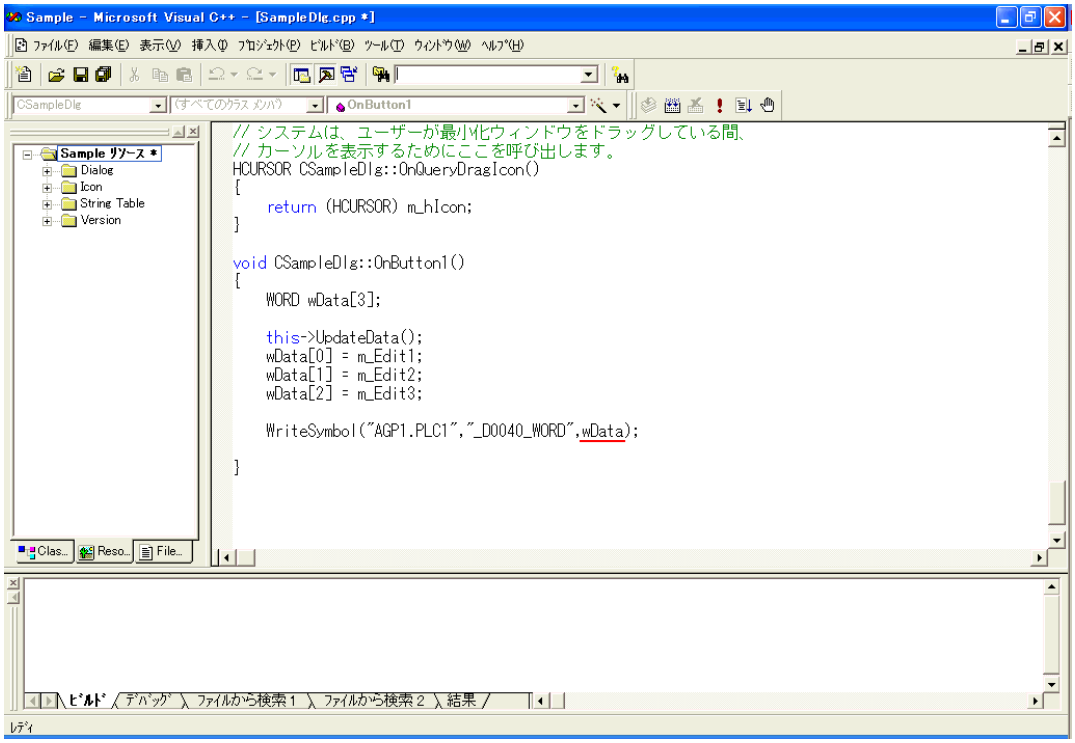
42 書き込みデータを格納するエリア（配列）を宣言します。書き込み点数が 3 点の場合、配列の要素数は 3 以上を指定してください。



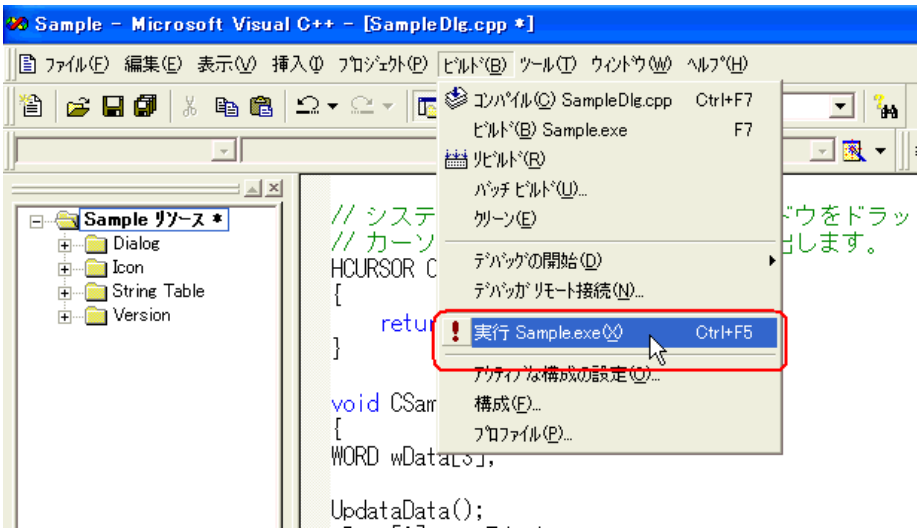
43 エディットボックスに入力された書き込みデータ（3 点分）を配列にセットします。



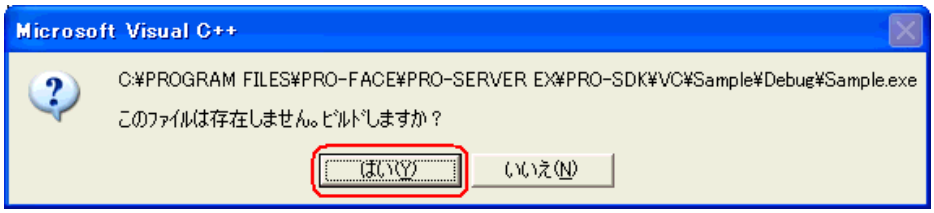
44 書き込みデータがセットされた配列の先頭（wData）を指定します。



45 Microsoft Visual C++ のメニューの［ビルド］から、［実行 Sample.exe］を選択します。



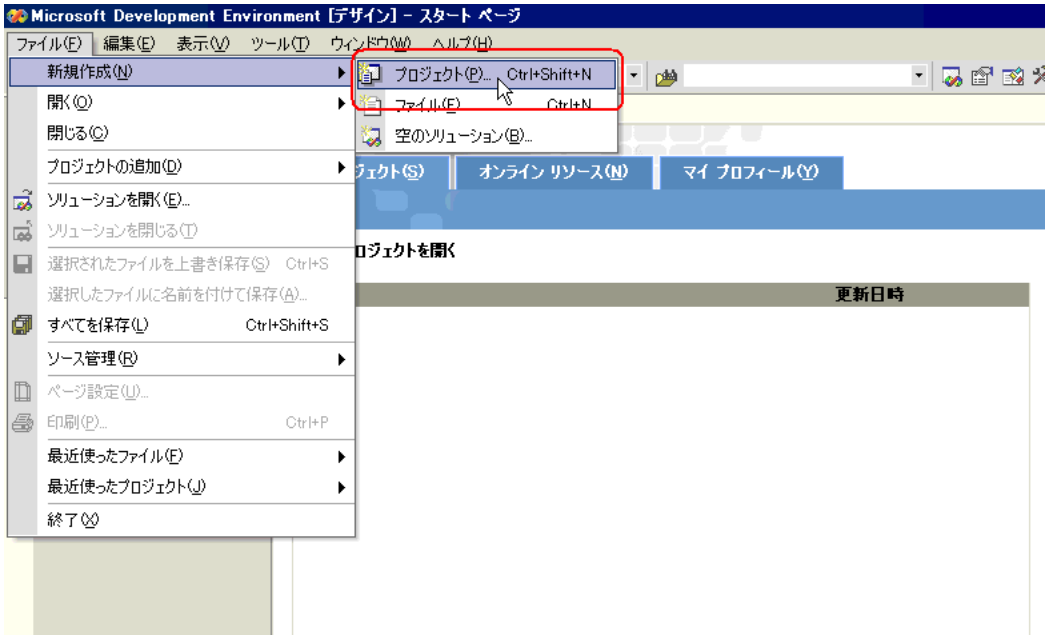
46 ［はい］ボタンをクリックします。



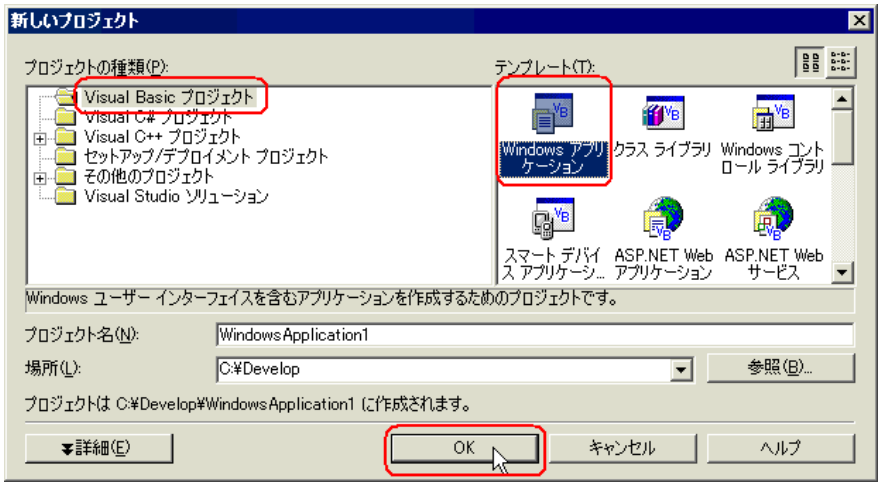
47 書き込む値（3 点分）を［エディットボックス］に入力した後、［Button1］をクリックすると、シンボル“ _D0040_WORD ”から 3 点分の書き込みが実行されます。

27.10.3 VB .NET 機能補助

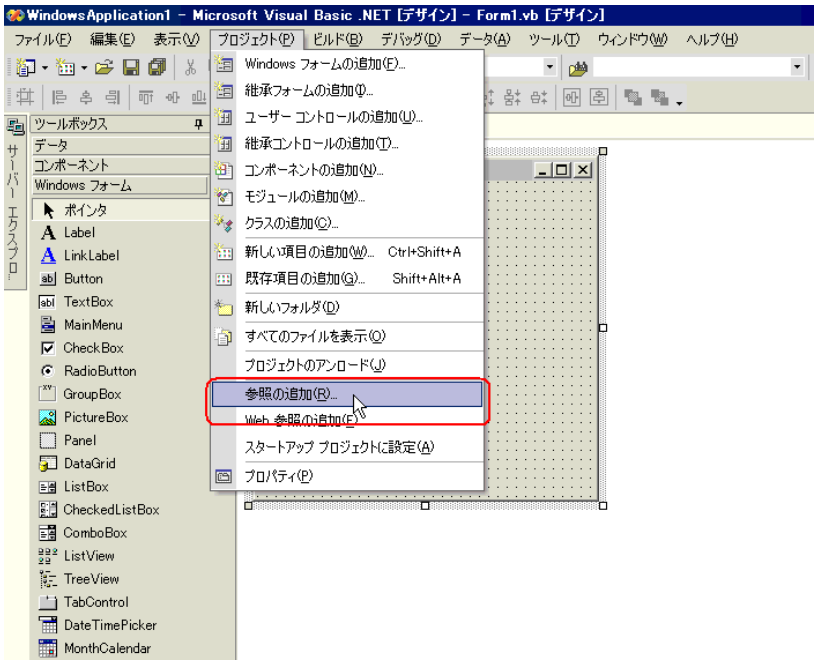
1 Microsoft Visual Studio .NET 2003（またはそれ以降）を起動し、メニューの［ファイル］から［新規作成］→［プロジェクト］を選択します。



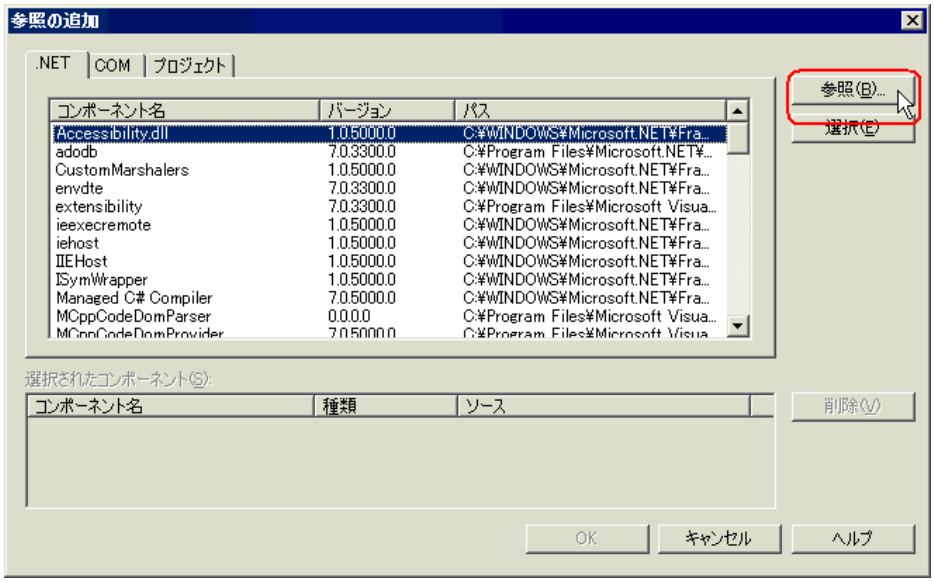
2［プロジェクトの種類］から［Visual Basic プロジェクト］を選択したあと、［テンプレート］から［Windows アプリケーション］を選択し、［OK］ボタンをクリックします。



3 メニューの［プロジェクト］から［参照の追加］を選択します。



4［参照］ボタンをクリックします。

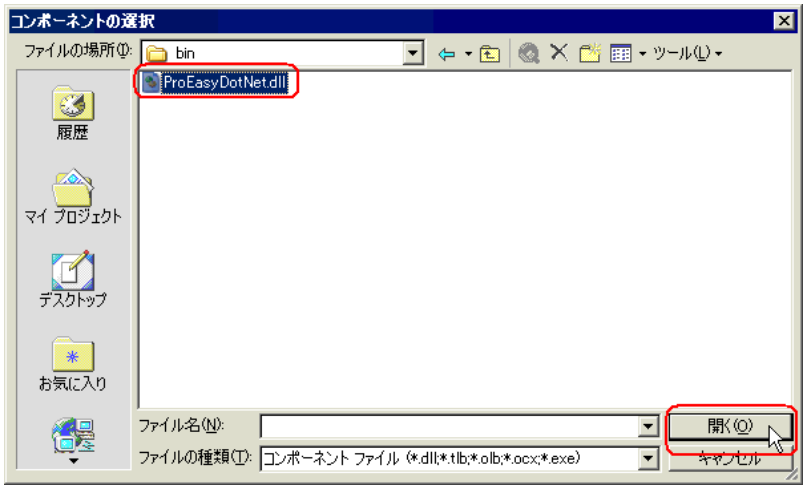


5 ProEazyDotNet.dll のインストール先ディレクトリを指定し、［開く］ボタンをクリックします。

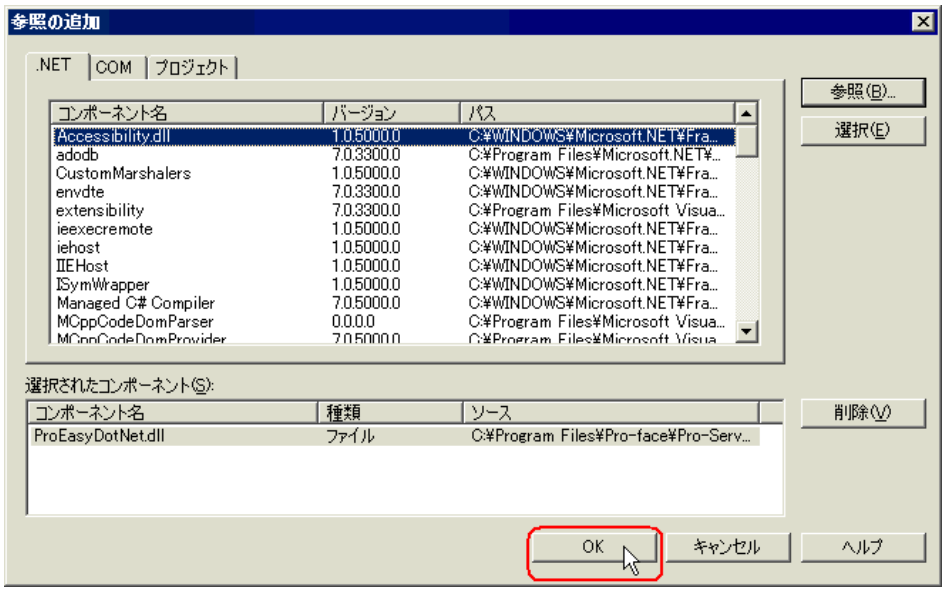
MEMO

・ 標準でインストールした場合は、以下の階層に位置します。

•Windows Vista：C:¥Pro-face¥Pro-Server EX¥PRO-SDK¥DotNet¥bin¥ProEazyDotNet.dll
•Windows Vista 以外
：C:¥Program Files¥Pro-face¥Pro-Server EX¥PRO-SDK¥DotNet¥bin¥ProEazyDotNet.dll



6［OK］ボタンをクリックします。



「ProEasyDotNet.dll」が登録されます。

以上で、VB .NET 使用の環境が整いました。
前記 1 ～ 6 の操作は、読み込み／書き込みのいずれの場合でも共通です。

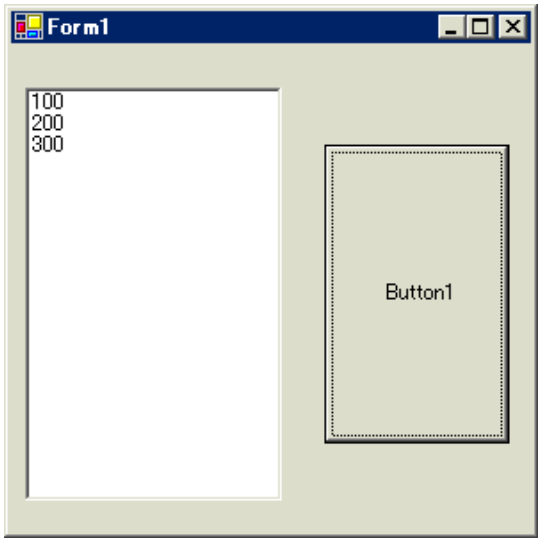
以降の手順については、読み込みの場合と書き込みの場合で手順が異なりますので、個別に説明します。

［読み込み］用アプリケーションの作成については、手順 7 ～ 19 をご覧ください。

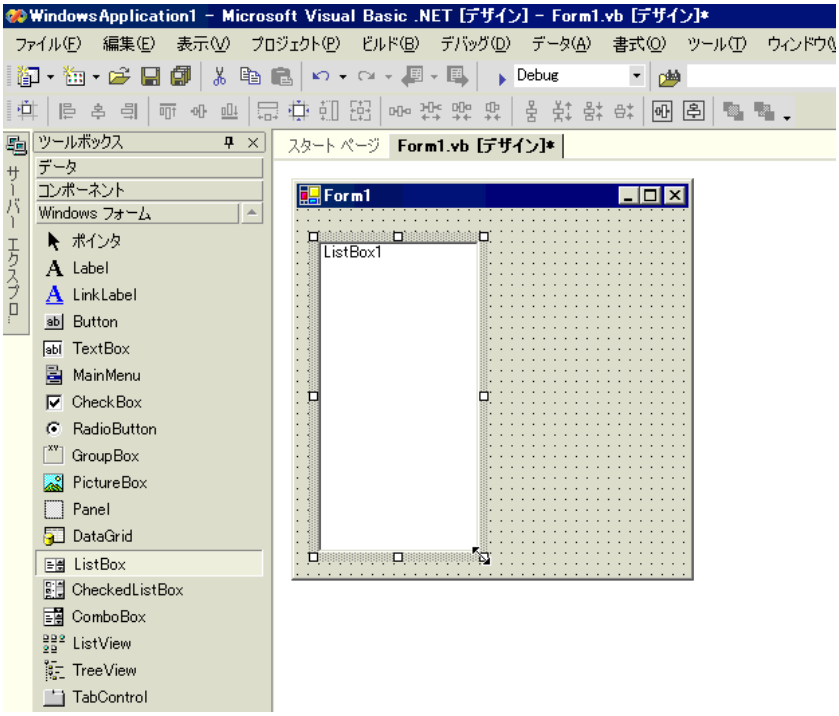
［書き込み］用アプリケーションの作成については、手順 20 ～ 32 をご覧ください。

〔読み込み〕用アプリケーションの作成

ここでは、［Button1］をクリックすると、3 点分のデータ（16 ビット符号付き）を読み出して表示するアプリケーションについて説明します。

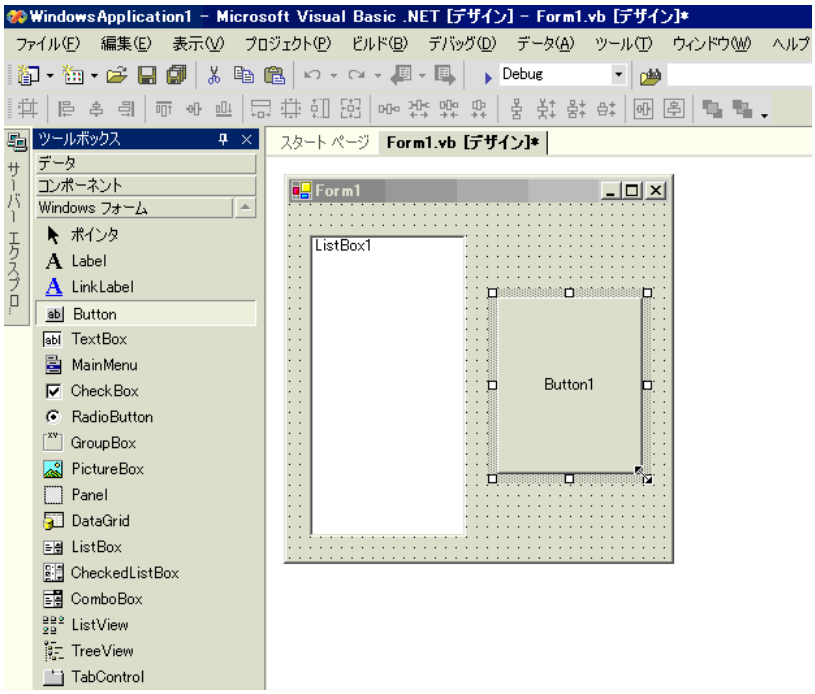


7［ツールボックス］内の［ListBox］を選択し、クリップして［Form1］に貼り付けます。

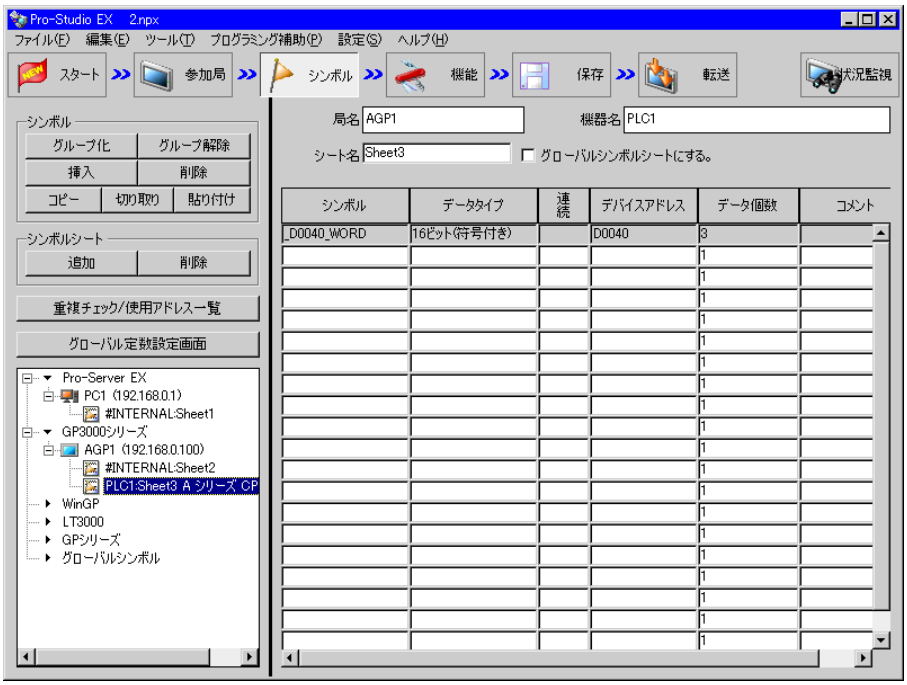


※　［ツールボックス］が表示されていない場合は、メニューの［表示］から［ツールボックス］を選択してください。

8 同様に、［Button］を選択し、クリップして［Form1］に貼り付けます。

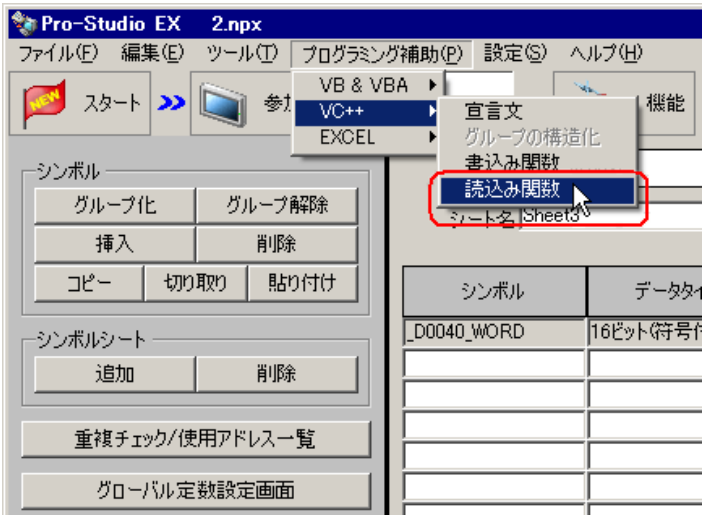


9『Pro-Studio EX』に登録されているシンボルで、読み込みたいシンボル名を選択します。

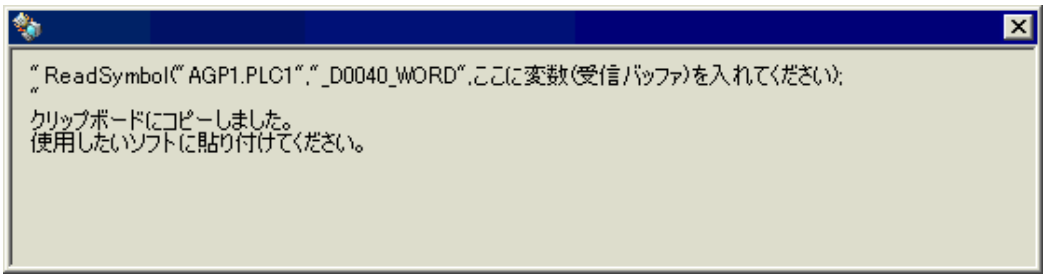


※ 本例では、データタイプが［16 ビット（符号付き）］で、データ個数が 3 個のシンボルを対象としています。

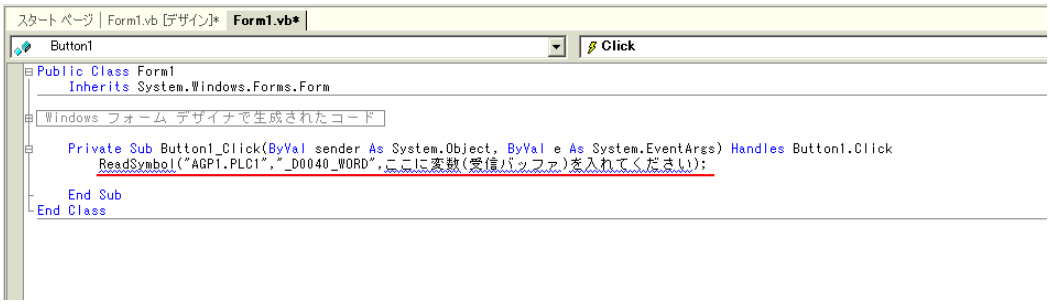
10 メニューの［プログラミング補助］から［VC++］→［読込み関数］を選択します。



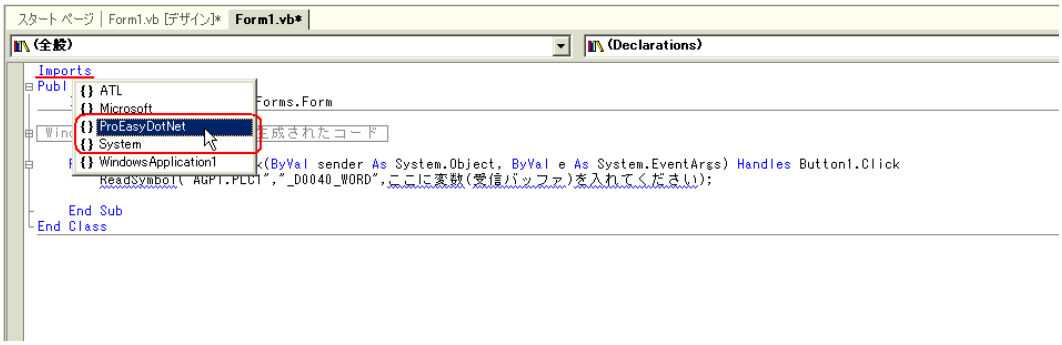
読み込み関数がクリップボードにコピーされます。



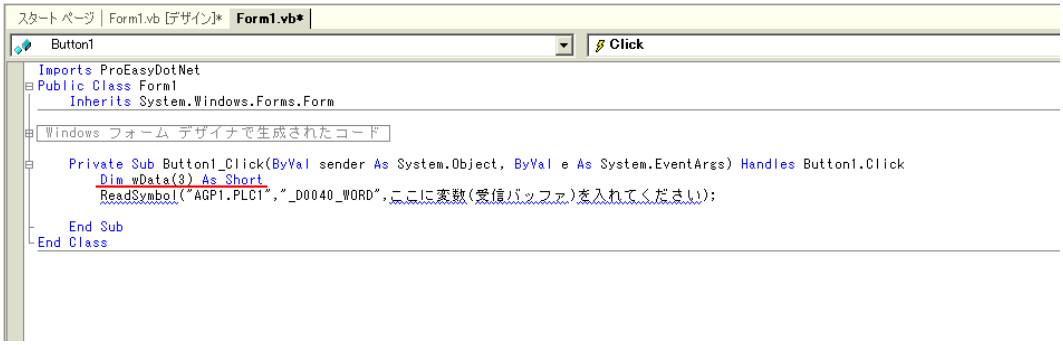
11［Form1］上の［Button1］をダブルクリックし、Sub ステートメントと End Sub ステートメントの間にクリップボードの内容（読み込み関数）を貼り付けます。



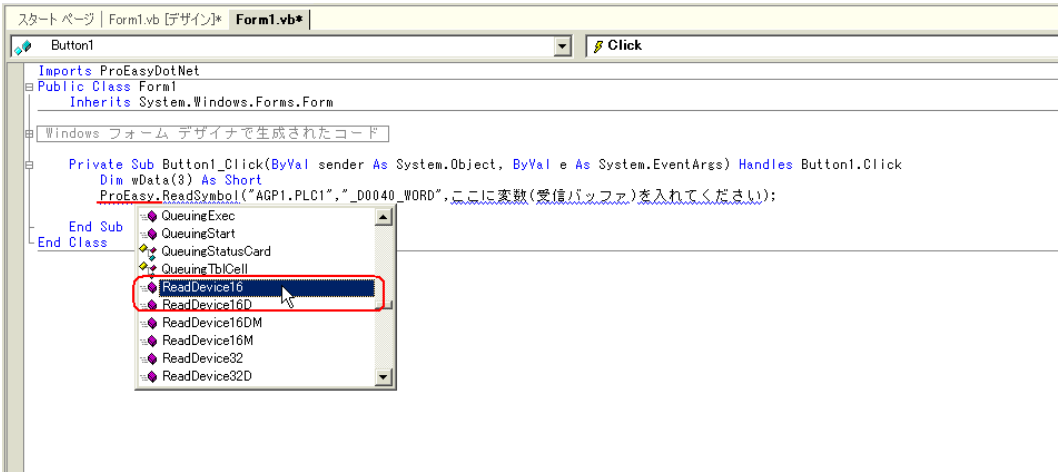
12 ProEasyDotNet のライブラリをインポートします。
ソースコードの先頭に “ Imports ” と入力し、表示されるリストボックスの中から［ProEasyDotNet］を選択します。



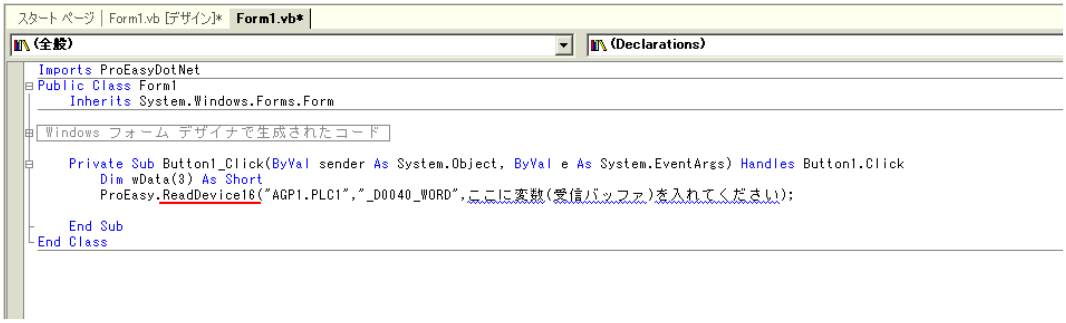
13 読み込んだデータを格納するエリアとして、変数 “ wData ” を宣言します。
配列の型（本例では “ Short ”）は、対象となるシンボルのデータタイプに合わせてください。長さは、対象となるシンボルと同じ長さ（本例では “ 3 ”）を指定します。



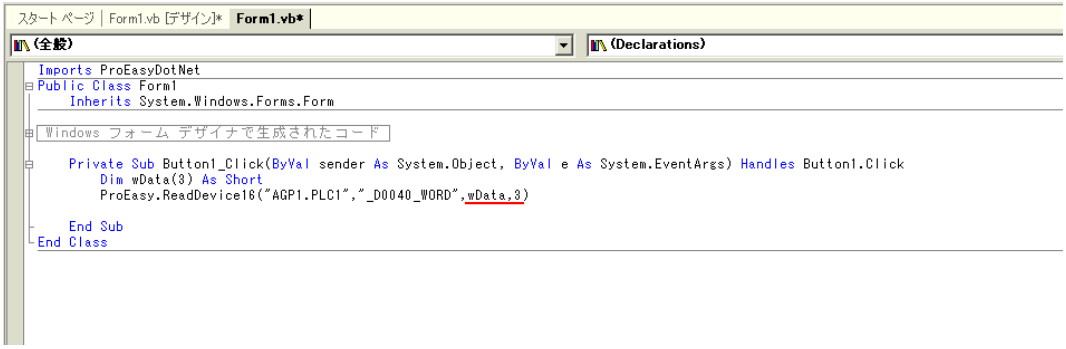
14 “ ReadSymbol ” の前に “ ProEasy. ” と入力し、表示されるリストボックスの中から［ReadDevice16］を選択します。



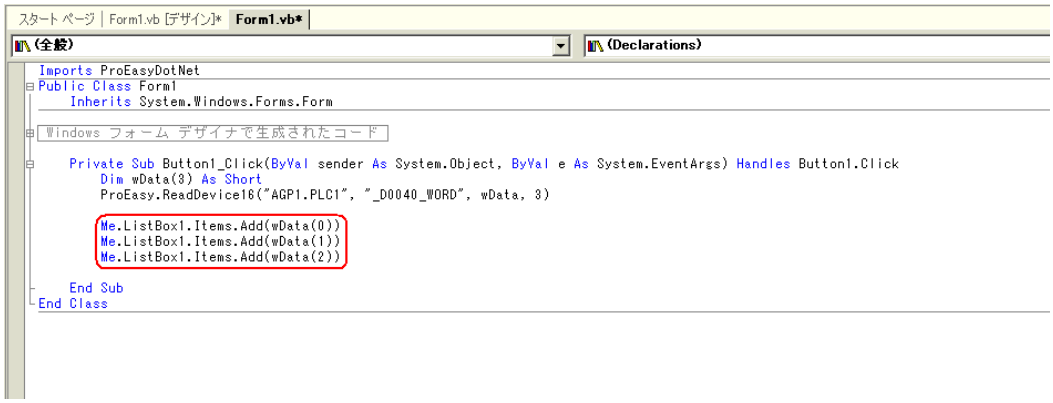
15 クリップボードから貼り付けた文字列（読み込み関数）の “ ReadSymbol ” を削除します。



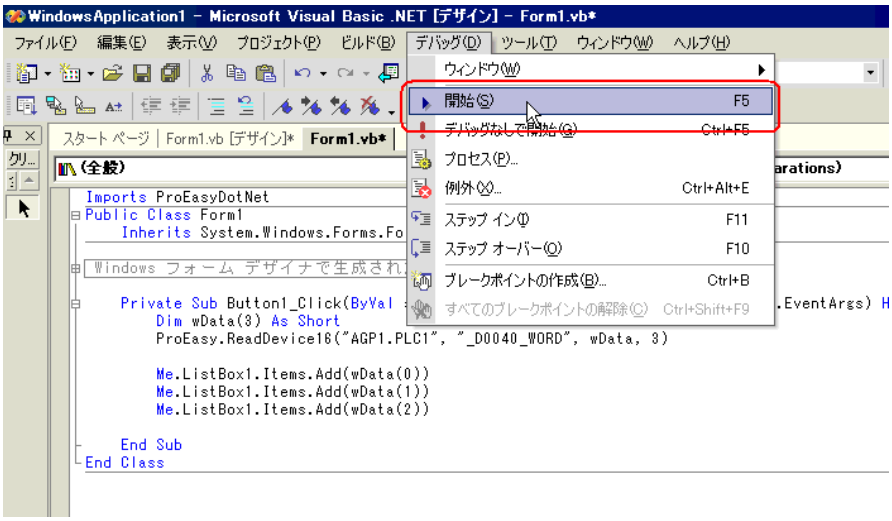
16 3 番目の引数として、データを格納するエリア “ wData ” を指定します。その後ろに “ , ”（カンマ）を入力し、4 番目の引数として、対象とするシンボルの長さ “ 3 ” を入力します。その後、行末の “ ; ”（セミコロン）を削除します。



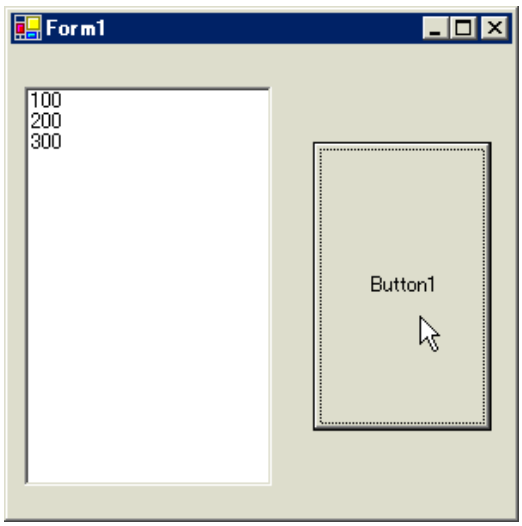
17 読み込んだデータ 3 点分（wData(0)、wData(1)、wData(2)）を［ListBox1］に順次追加します。



18 メニューの［デバッグ］から［開始］を選択します。

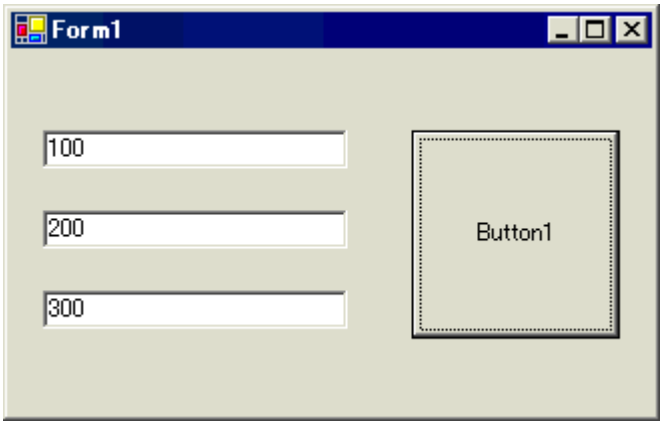


19［Button1］をクリックすると、対象シンボルのデータ（3 点）が［ListBox］に表示されます。

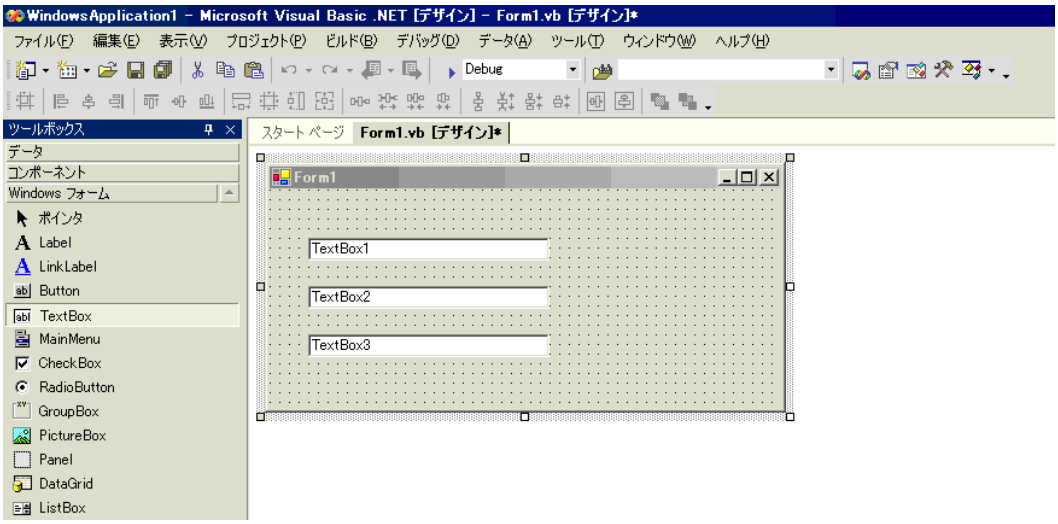


〔書き込み〕用アプリケーションの作成

ここでは、［Button1］をクリックすると、3 点分のデータ（16 ビット符号付き）を書き込むアプリケーションについて説明します。

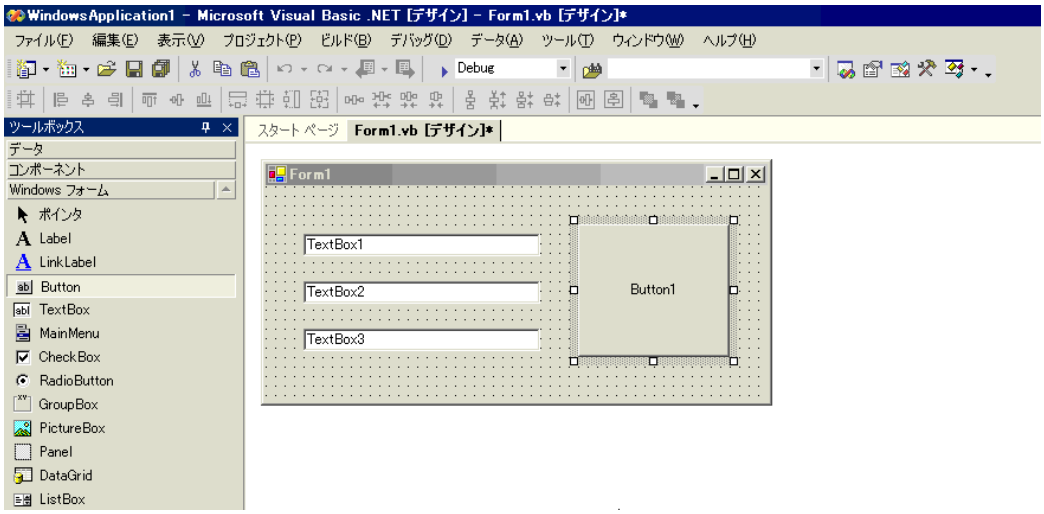


20 ［ツールボックス］内の［TextBox］を選択し、クリップして［Form1］に 3 個貼り付けます。



※　［ツールボックス］が表示されていない場合は、メニューの［表示］から［ツールボックス］をクリックしてください。

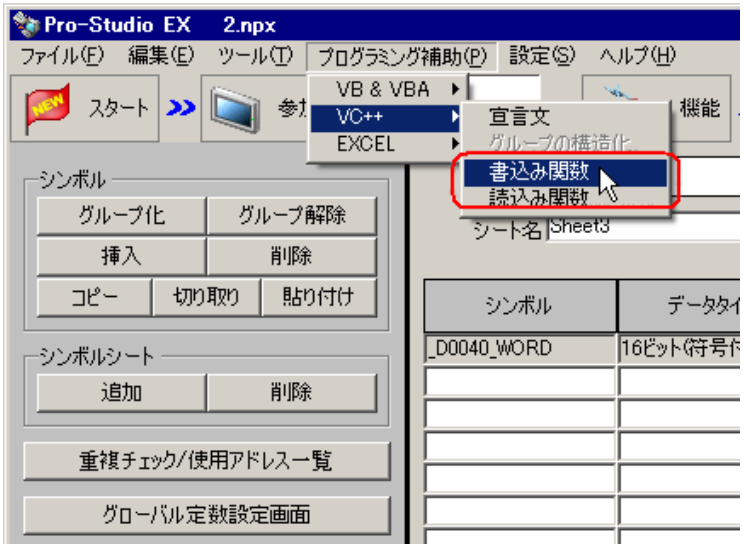
21 同様に、［Button］を選択し、クリップして［Form1］に貼り付けます。



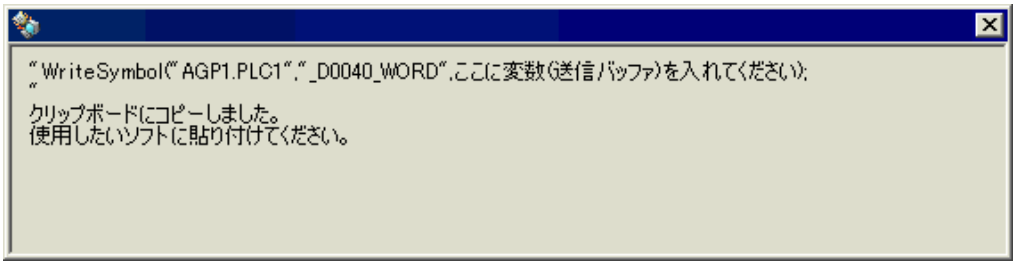
22『Pro-Studio EX』に登録されているシンボルで、書き込み先のシンボル名を選択します。（書き込み先の先頭を選択してください。）

※ 本例では、データタイプが［16 ビット（符号付き）］で、データ個数が 3 個のシンボルを対象としています。

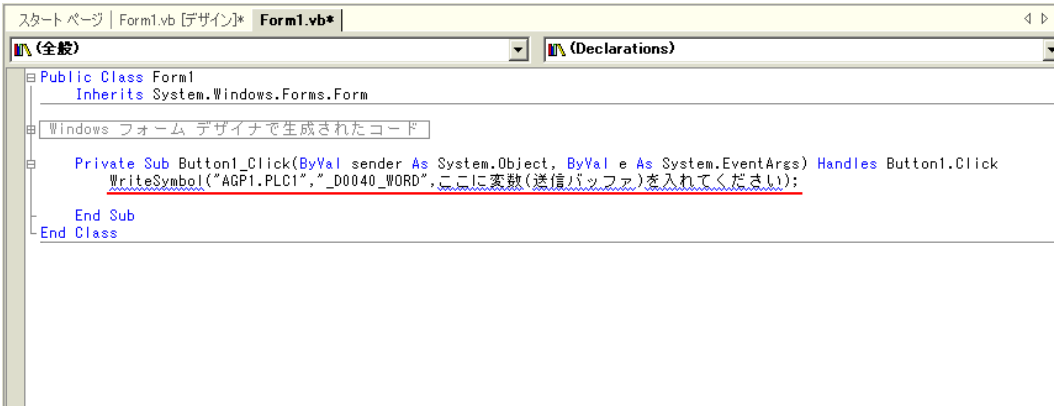
23 メニューの［プログラミング補助］から［VC++］→［書込み関数］を選択します。



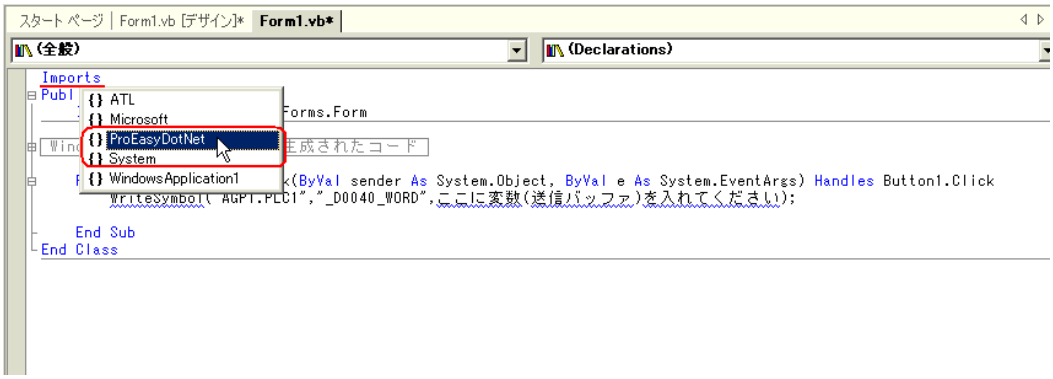
書き込み関数がクリップボードにコピーされます。



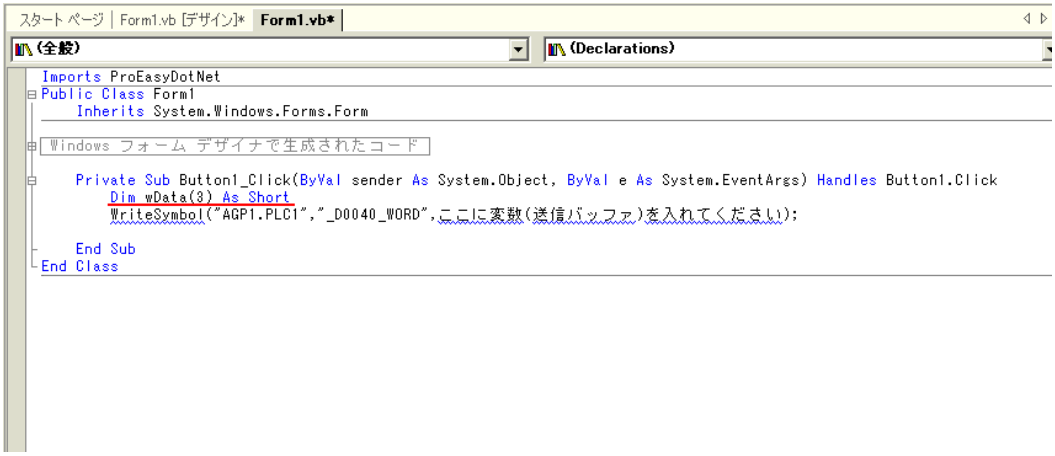
24 ［Form1］上の［Button1］をダブルクリックし、［Button1_Click］メソッド（文字列 " Private Sub Button1_Click... "）の下にクリップボードの内容（書き込み関数）を貼り付けます。



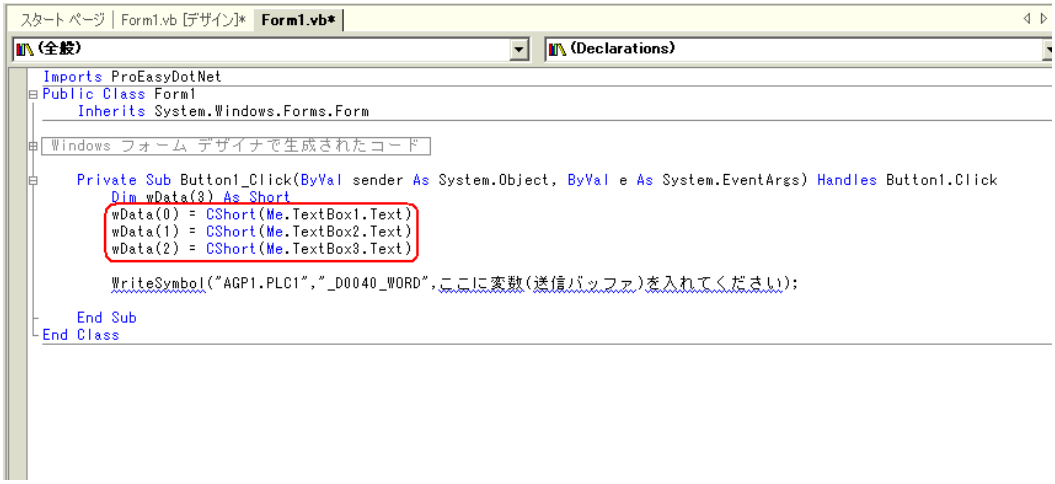
25 ProEasyDotNet のライブラリをインポートします。
ソースコードの先頭に “ Imports ” と入力し、表示されるリストボックスの中から［ProEasyDotNet］を選択します。



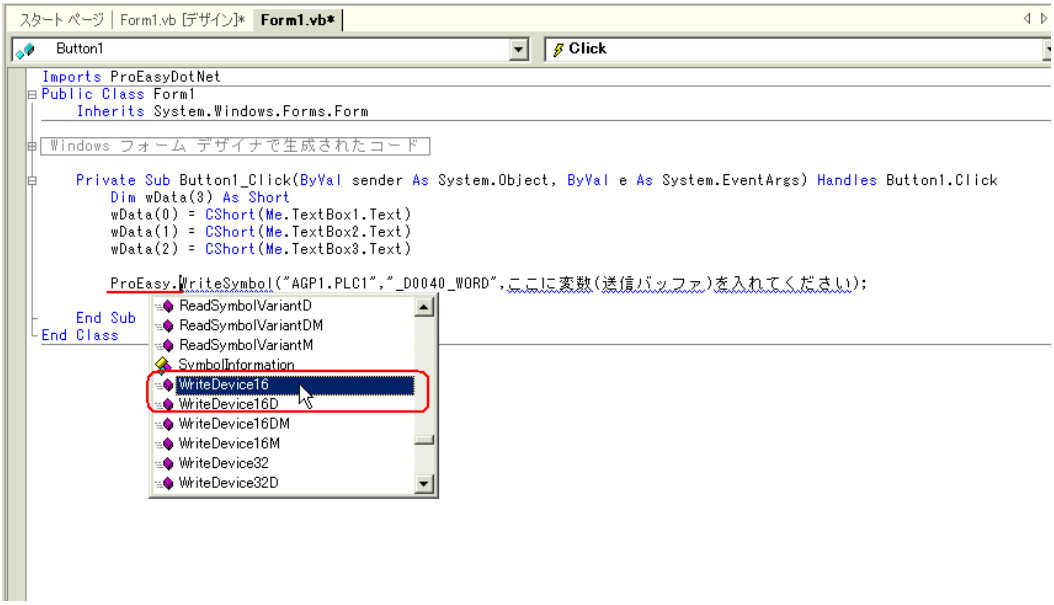
26 書き込むデータを格納するエリアとして、変数 “ wData ” を宣言します。
配列の型（本例では “ Short ”）は、対象となるシンボルのデータタイプに合わせてください。長さは、対象となるシンボルと同じ長さ（本例では “ 3 ”）を指定します。



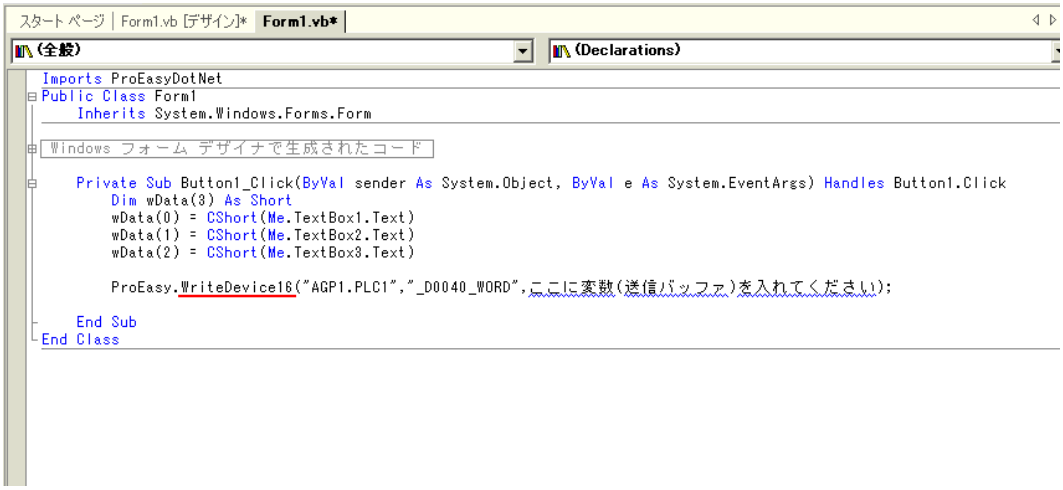
27［TextBox1］～［TextBox3］に入力するデータを、配列にセットします。



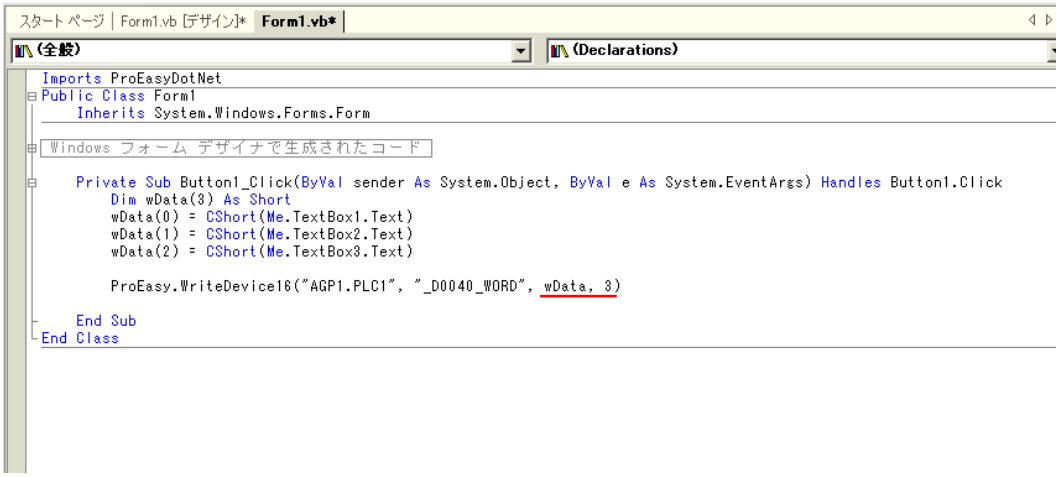
28“ WriteSymbol ”の前に“ ProEasy. ”と入力し、表示されるリストボックスの中から［WriteDevice16］を選択します。



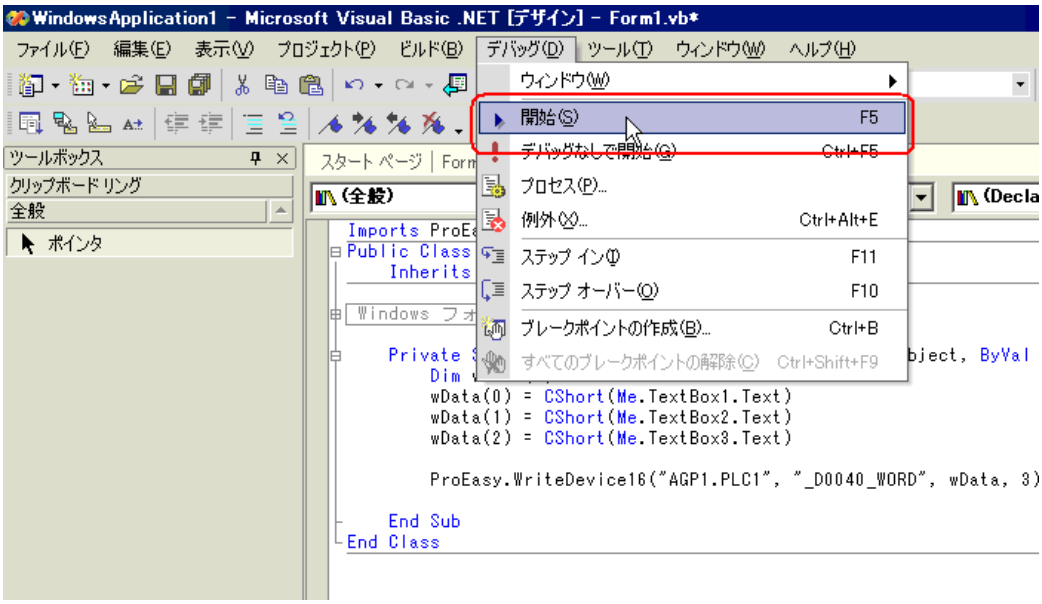
29 クリップボードから貼り付けた文字列（書き込み関数）の “ WriteSymbol ” を削除します。



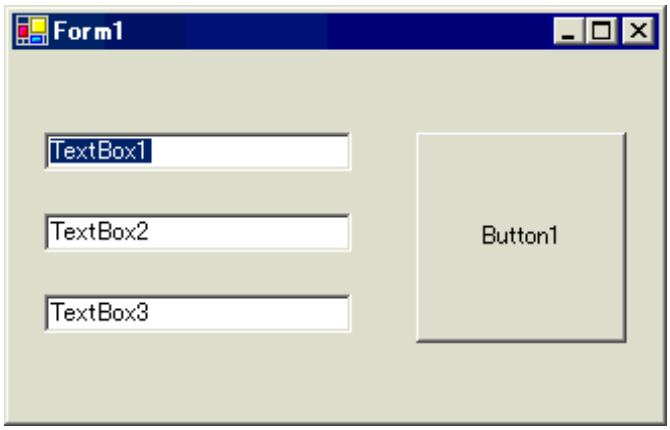
30 3 番目の引数として、データを格納するエリア “ wData ” を指定します。その後ろに “ , ”（カンマ）を入力し、4 番目の引数として、対象とするシンボルの長さ “ 3 ” を入力します。その後、行末の “ ; ”（セミコロン）を削除します。



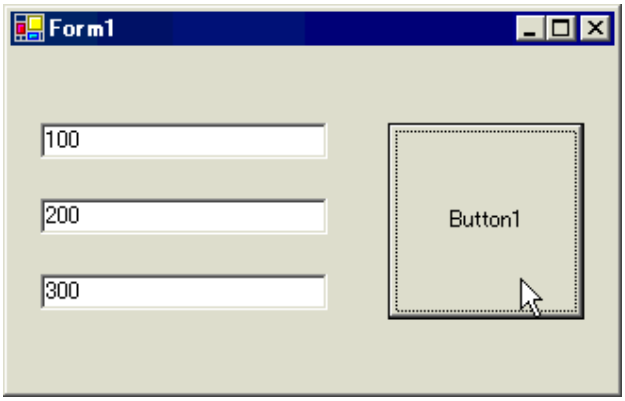
31 メニューの［デバッグ］から［開始］を選択します。



32 起動直後には、［TextBox］に文字列 “ TextBox* ” が表示されています。

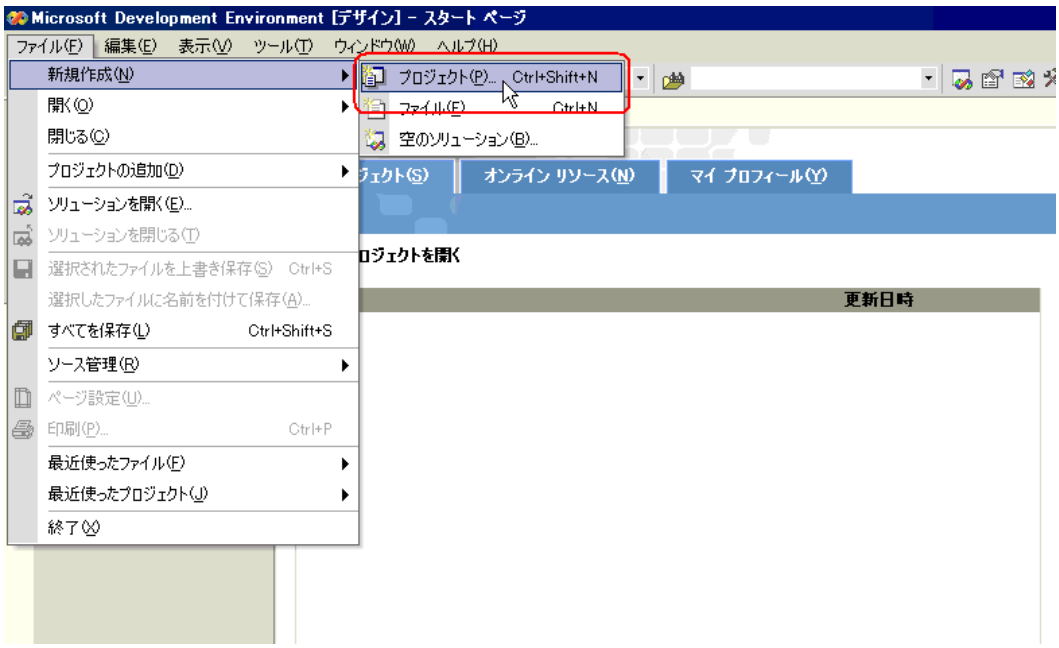


書き込むデータ（3 点分）を［TextBox］に入力したあと、［Button1］をクリックすると、データがシンボルで指定した箇所に書き込まれます。

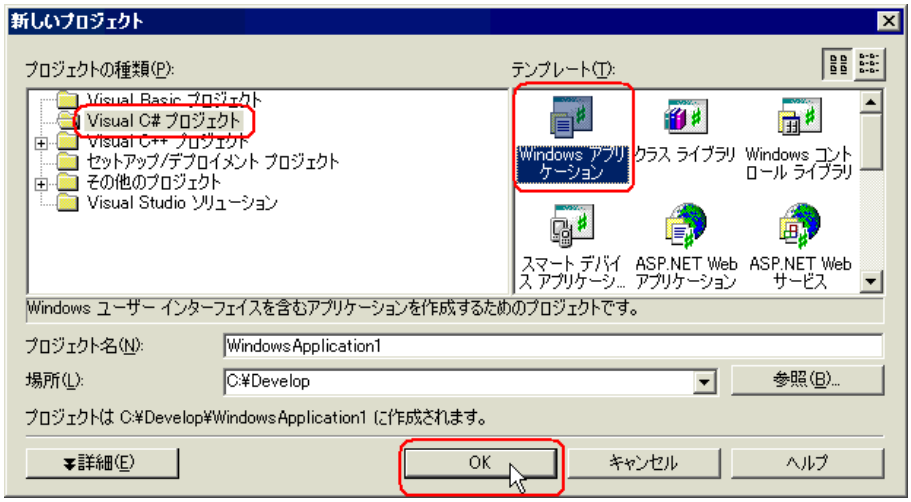


27.10.4 C# .NET 機能補助

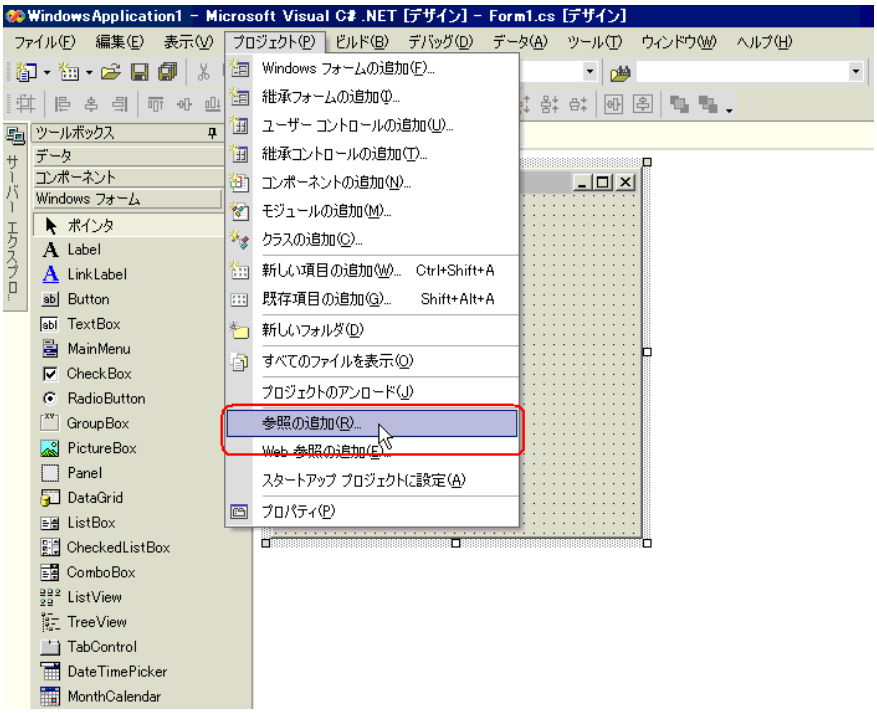
1 Microsoft Visual Studio .NET 2003（またはそれ以降）を起動し、メニューの［ファイル］から［新規作成］→［プロジェクト］を選択します。



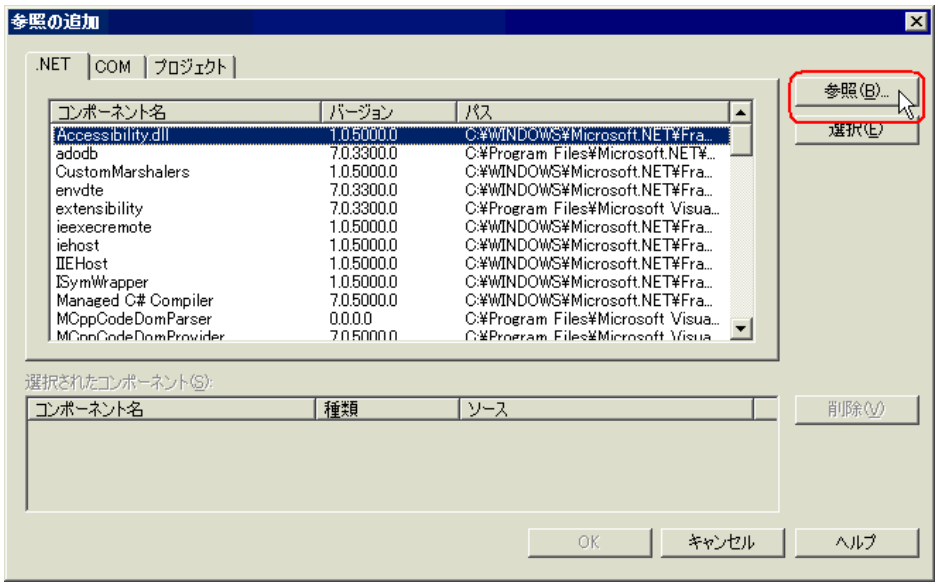
2［プロジェクトの種類］から［Visual C# プロジェクト］を選択したあと、［テンプレート］から［Windows アプリケーション］を選択し、［OK］ボタンをクリックします。



3 メニューの［プロジェクト］から［参照の追加］を選択します。

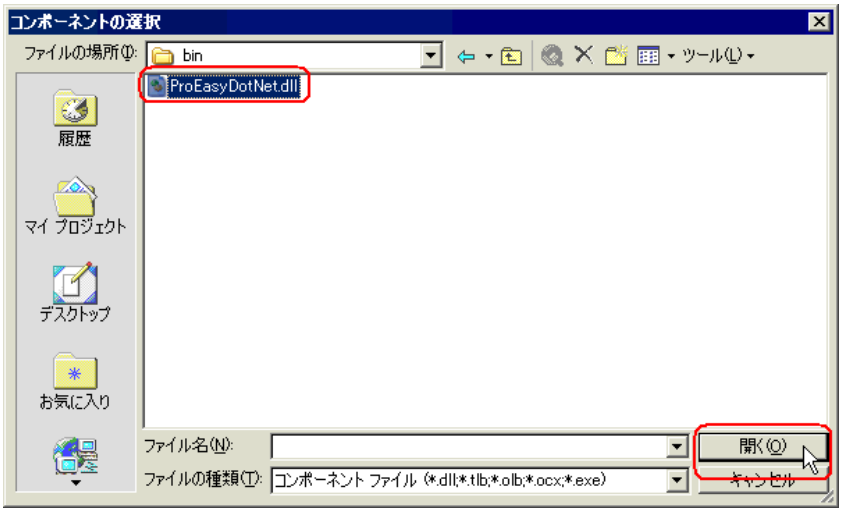


4［参照］ボタンをクリックします。

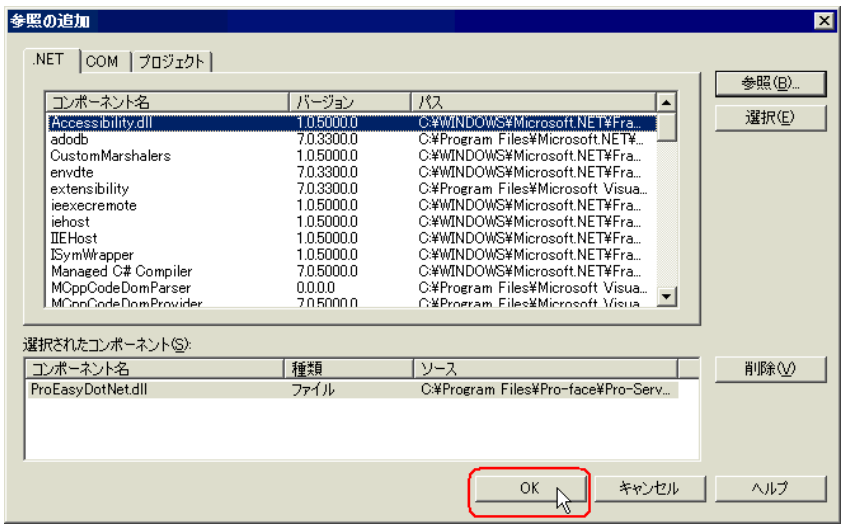


5 ProEazyDotNet.dll のインストール先ディレクトリを指定し、［開く］ボタンをクリックします。

MEMO
- 標準でインストールした場合は、以下の階層に位置します。
  - •Windows Vista：C:¥Pro-face¥Pro-Server EX¥PRO-SDK¥DotNet¥bin¥ProEazyDotNet.dll
  - •Windows Vista 以外
  - ：C:¥Program Files¥Pro-face¥Pro-Server EX¥PRO-SDK¥DotNet¥bin¥ProEazyDotNet.dll



6［OK］ボタンをクリックします。



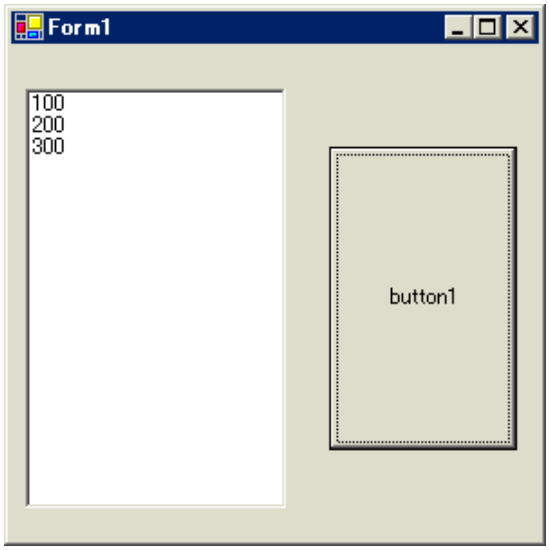
「ProEasyDotNet.dll」が登録されます。

以上で、C# .NET 使用の環境が整いました。
前記 1 ～ 6 の操作は、読み込み／書き込みのいずれの場合でも共通です。
以降の手順については、読み込みの場合と書き込みの場合で手順が異なりますので、個別に説明します。

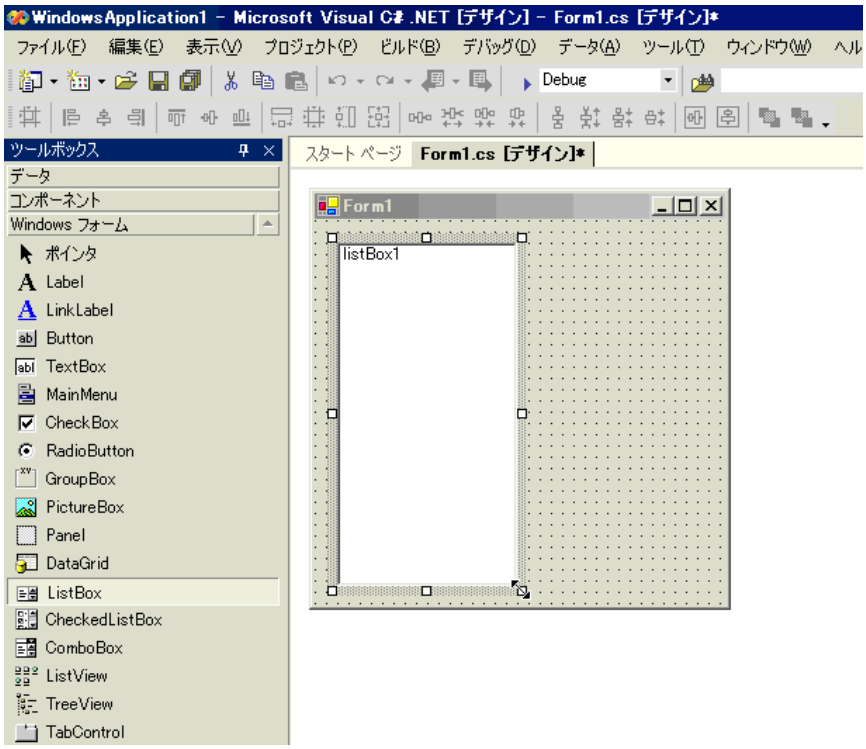
［読み込み］用アプリケーションの作成については、手順 7 ～ 19 をご覧ください。
［書き込み］用アプリケーションの作成については、手順 20 ～ 32 をご覧ください。

〔読み込み〕用アプリケーションの作成

ここでは、［button1］をクリックすると、3 点分のデータ（16 ビット符号付き）を読み出して表示するアプリケーションについて説明します。

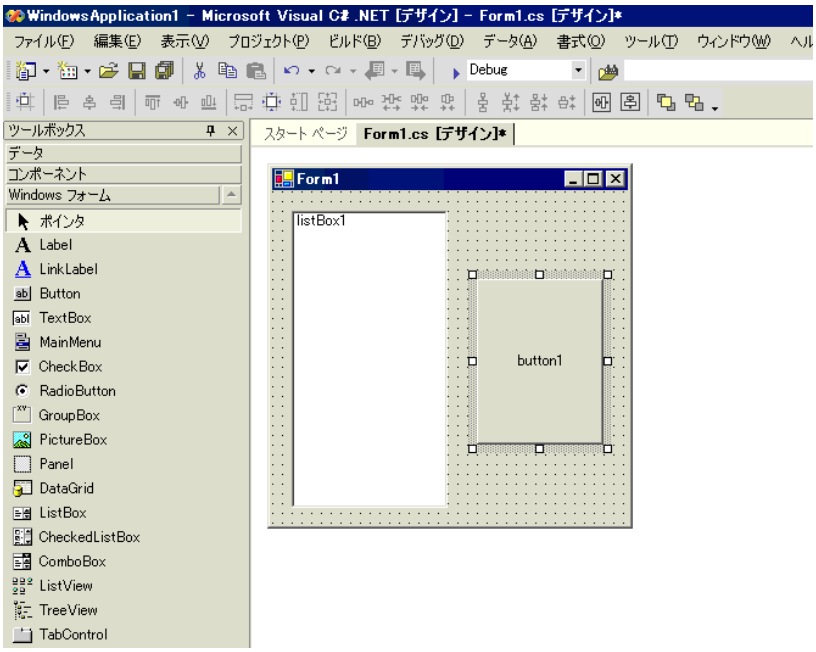


7［ツールボックス］内の［ListBox］を選択し、クリップして［Form1］に貼り付けます。

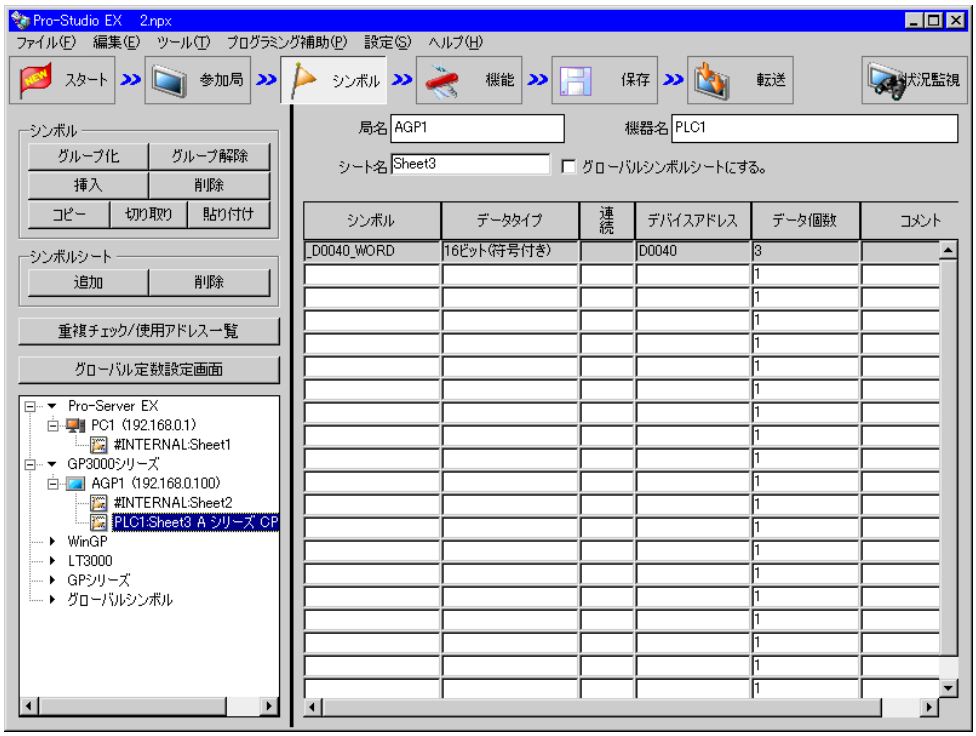


※　［ツールボックス］が表示されていない場合は、メニューの［表示］から［ツールボックス］を選択してください。

8 同様に、［Button］を選択し、クリップして［Form1］に貼り付けます。

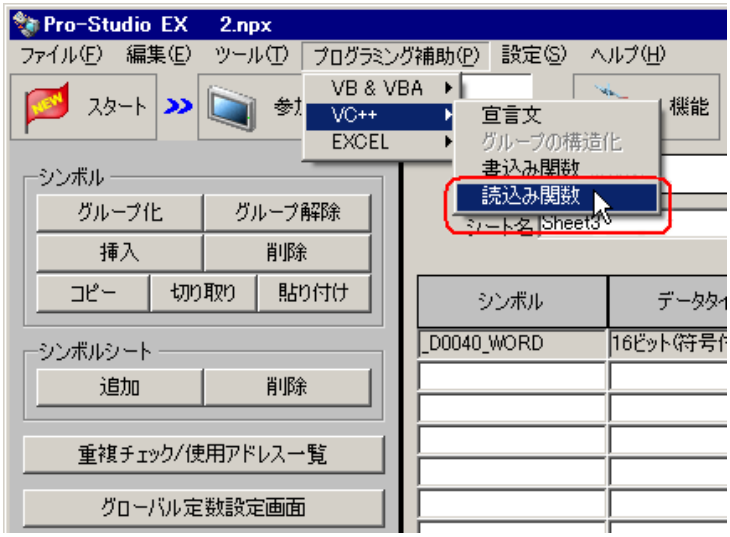


9『Pro-Studio EX』に登録されているシンボルで、読み込みたいシンボル名を選択します。

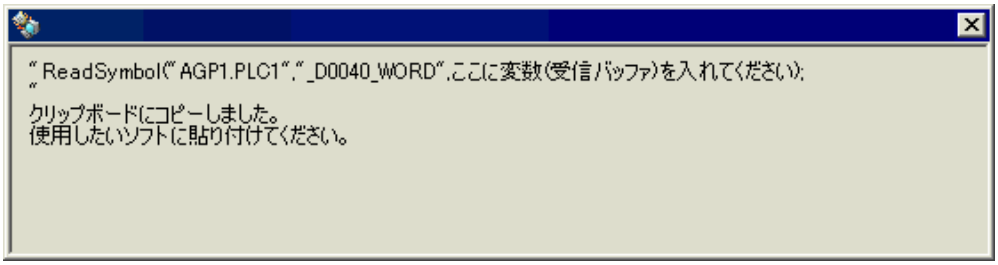


※ 本例では、データタイプが［16 ビット（符号付き）］で、データ個数が 3 個のシンボルを対象としています。

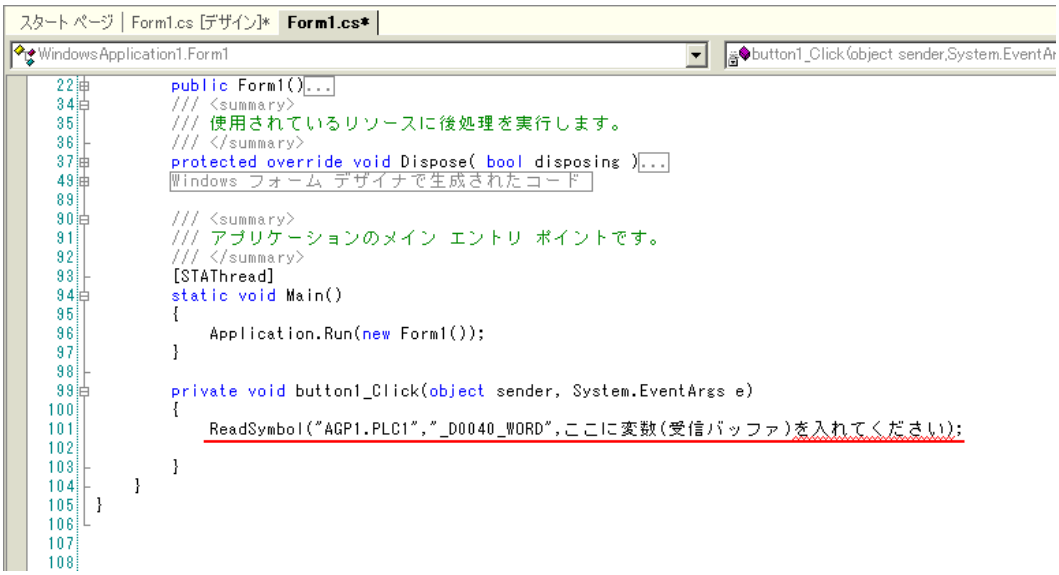
10 メニューの［プログラミング補助］から［VC++］→［読込み関数］を選択します。



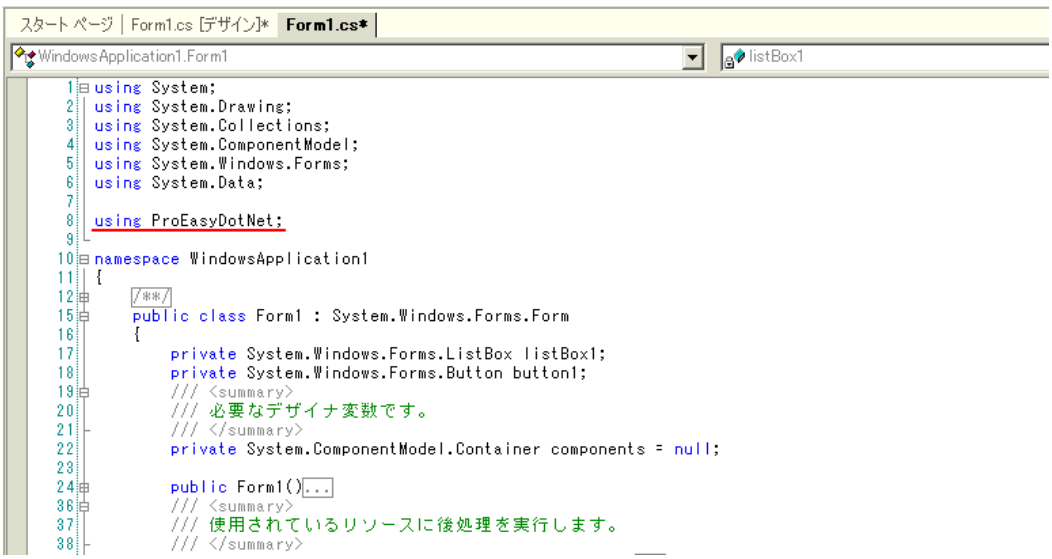
読み込み関数がクリップボードにコピーされます。



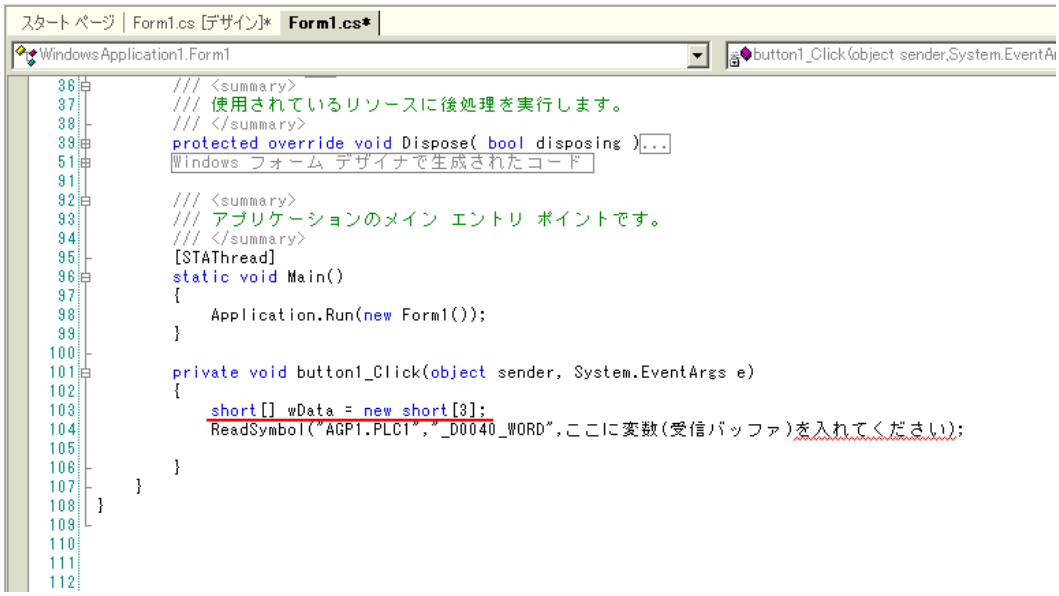
11 ［Form1］上の［button1］をダブルクリックし、［button1_Click］メソッド（文字列 “ private void button1_Click... ”）の下にクリップボードの内容（読み込み関数）を貼り付けます。



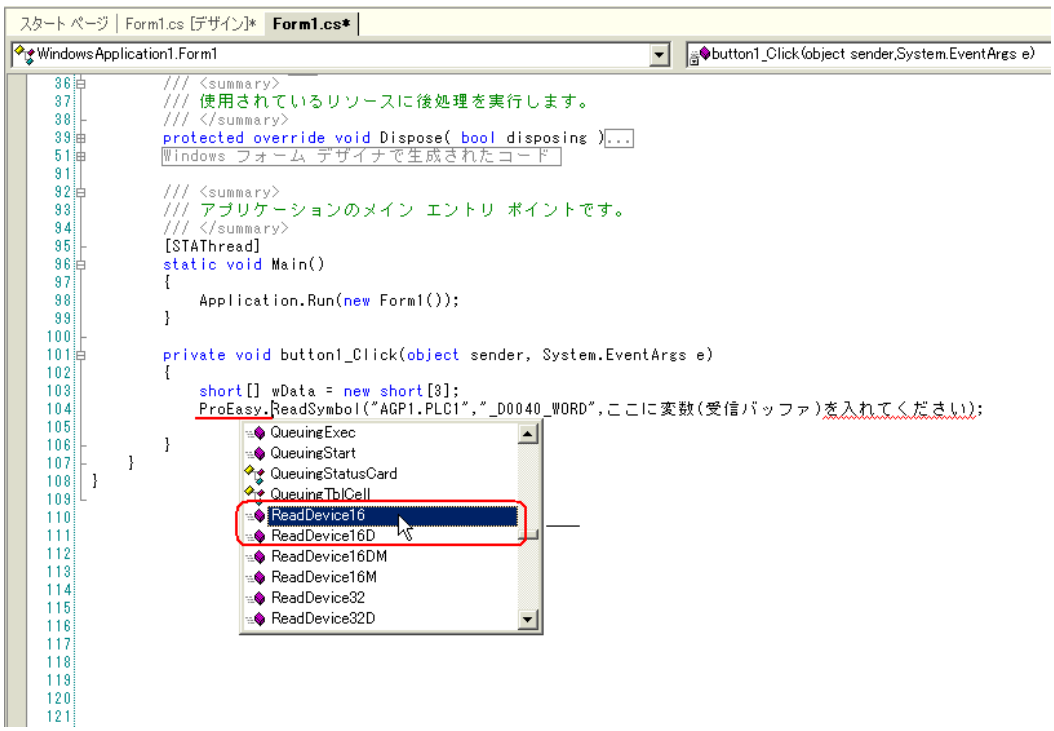
12 ProEasyDotNet のディレクティブを記述します。
ソースコードの先頭に存在する “ using... ” が記述されている部分の一番下に、“ using ProEasyDotNet; ” と入力します。



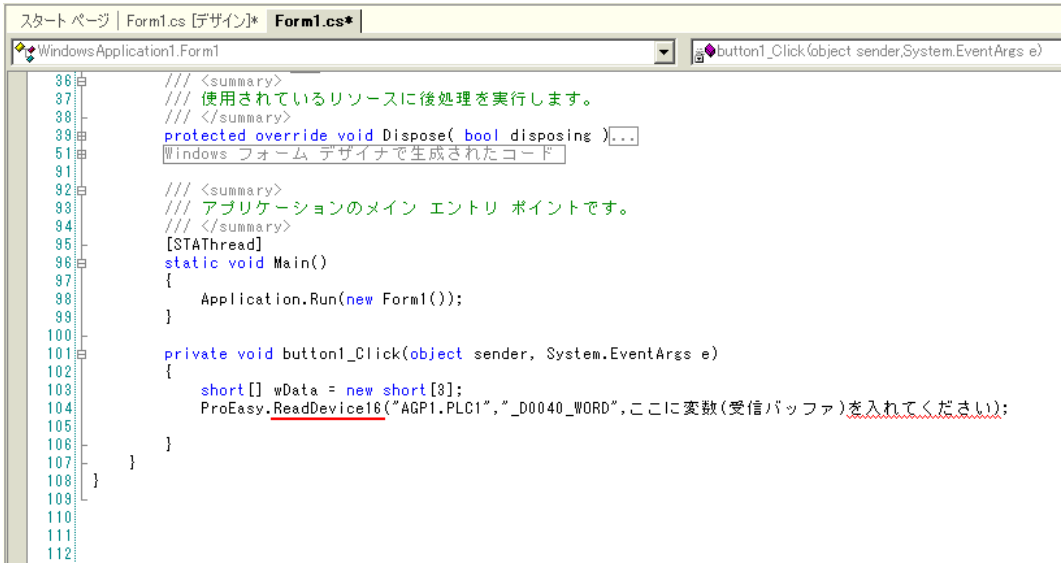
13 読み込んだデータを格納するエリアとして、変数“ wData ”を宣言します。
配列の型（本例では“ Short ”）は、対象となるシンボルのデータタイプに合わせてください。長さは、対象となるシンボルと同じ長さ（本例では“ 3 ”）を指定します。



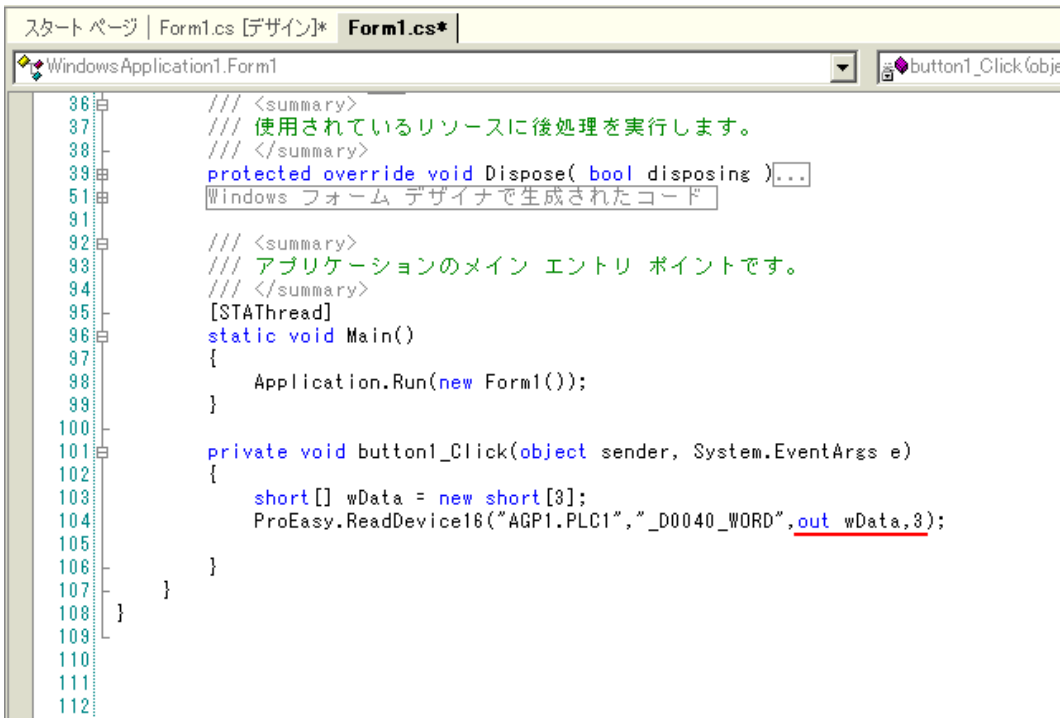
14“ ReadSymbol ”の前に“ ProEasy. ”と入力し、表示されるリストボックスの中から［ReadDevice16］を選択します。



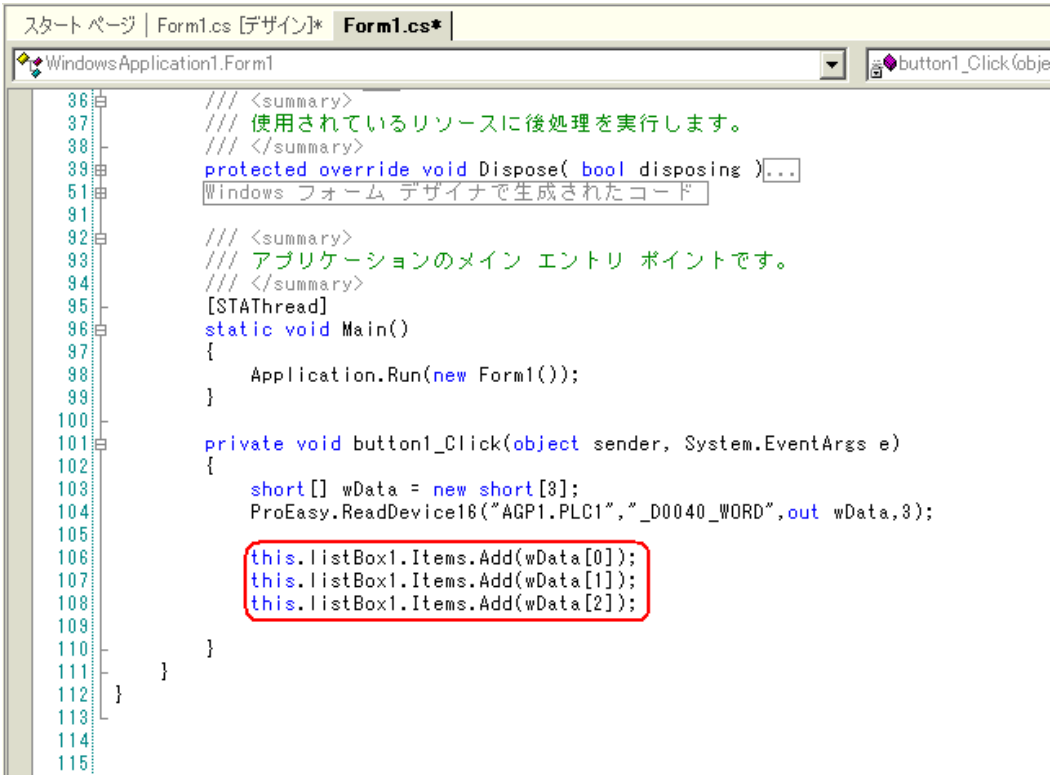
15 クリップボードから貼り付けた文字列（読み込み関数）の “ ReadSymbol ” を削除します。



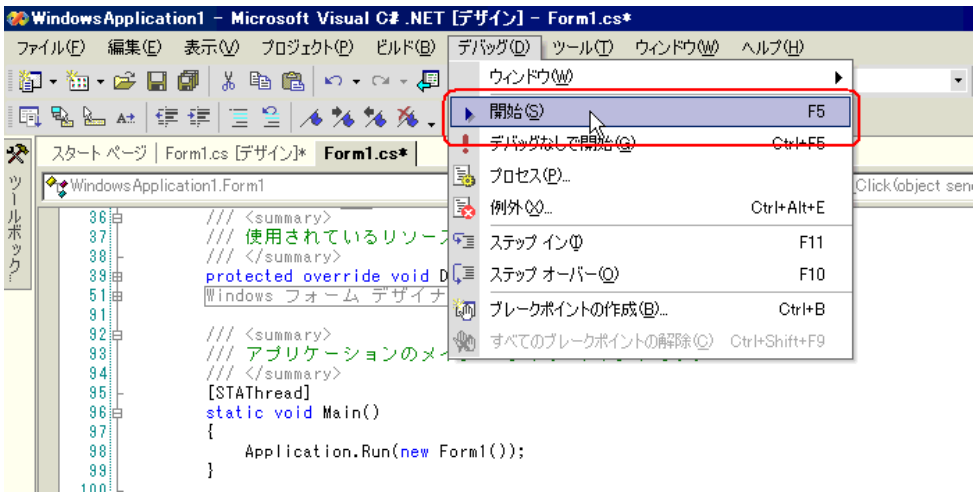
16 3 番目の引数として、データを格納するエリア “ wData ” を参照修飾子（out）付きで指定します。その後ろに “ , ”（カンマ）を入力し、4 番目の引数として、対象とするシンボルの長さ “ 3 ” を入力します。



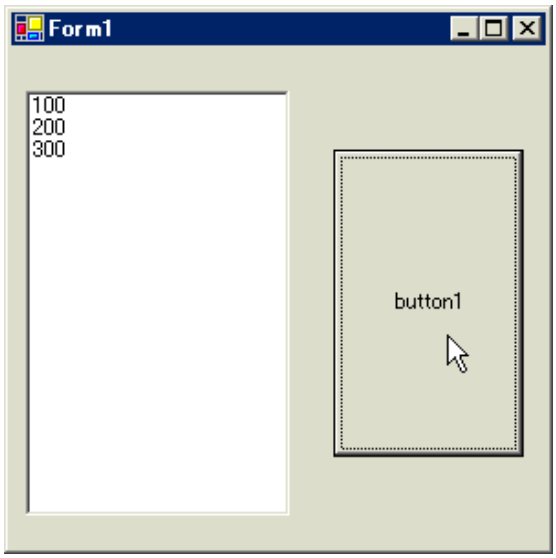
17 読み込んだデータ 3 点分（wData[0]、wData[1]、wData[2]）を［listBox1］に順次追加します。



18 メニューの［デバッグ］から［開始］を選択します。

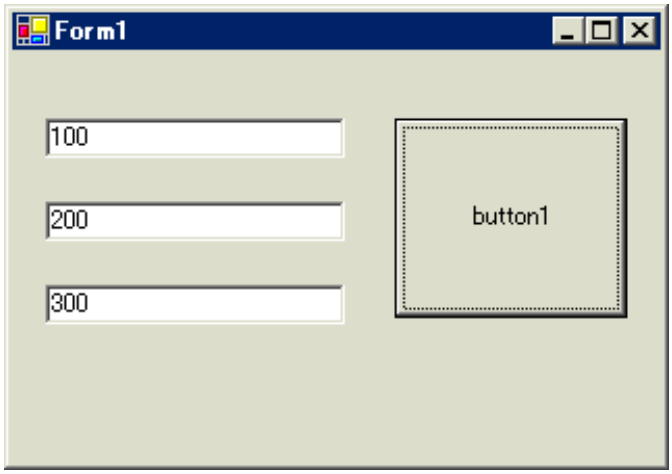


19［button1］をクリックすると、対象シンボルのデータ（3 点）が［ListBox］に表示されます。

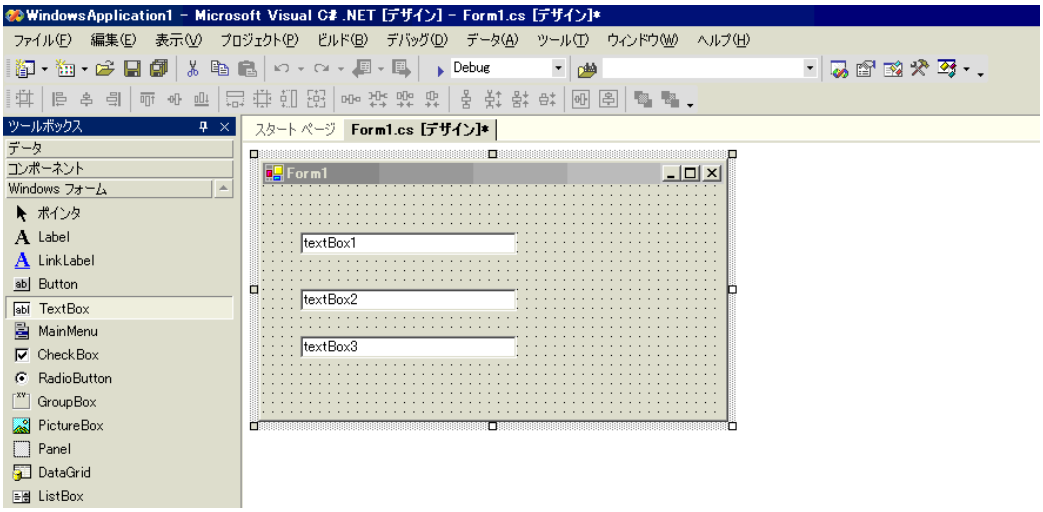


〔書き込み〕用アプリケーションの作成

ここでは、［button1］をクリックすると、3 点分のデータ（16 ビット符号付き）を書き込むアプリケーションについて説明します。

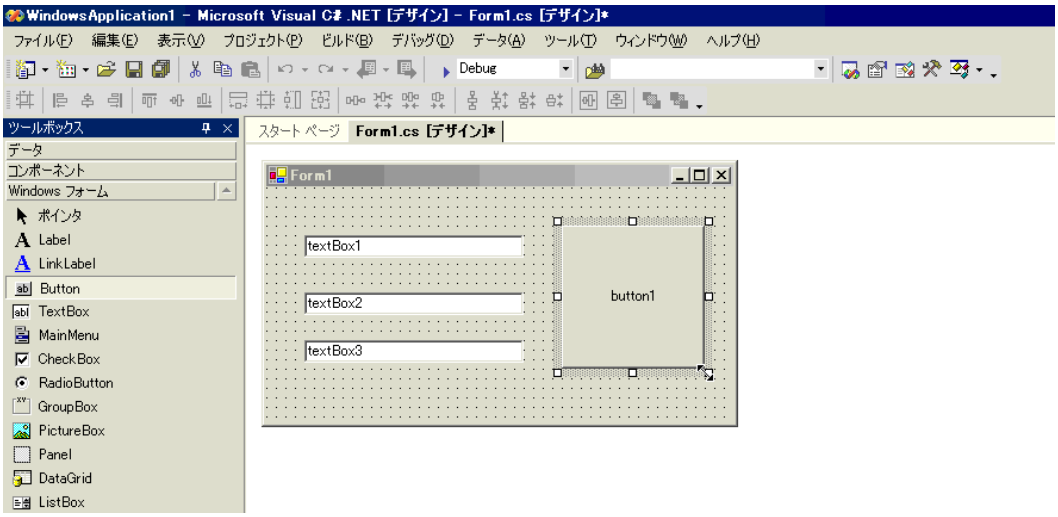


20 ［ツールボックス］内の［TextBox］を選択し、クリップして［Form1］に 3 個貼り付けます。



※　［ツールボックス］が表示されていない場合は、メニューの［表示］から［ツールボックス］をクリックしてください。

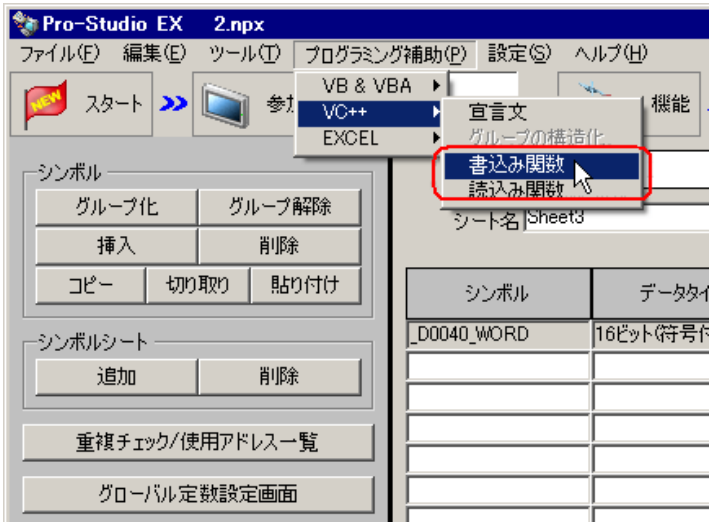
21 同様に、［Button］を選択し、クリップして［Form1］に貼り付けます。



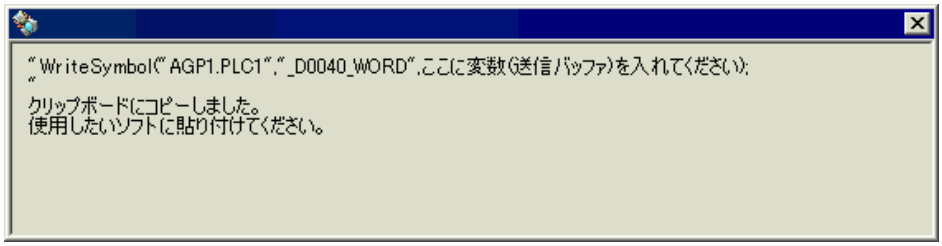
22『Pro-Studio EX』に登録されているシンボルで、書き込み先のシンボル名を選択します。（書き込み先の先頭を選択してください。）

※ 本例では、データタイプが［16 ビット（符号付き）］で、データ個数が 3 個のシンボルを対象としています。

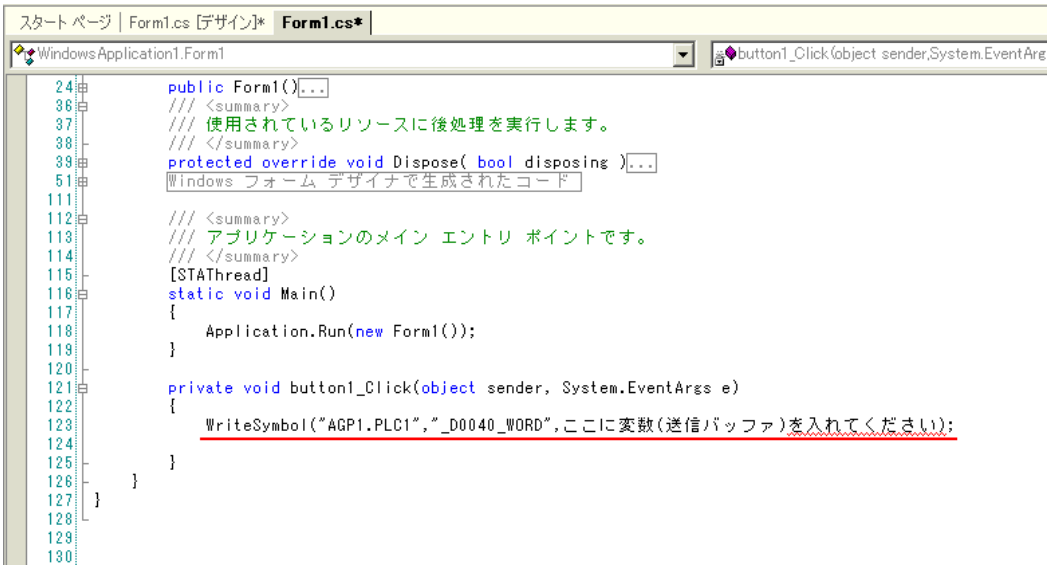
23 メニューの［プログラミング補助］から［VC++］→［書込み関数］を選択します。



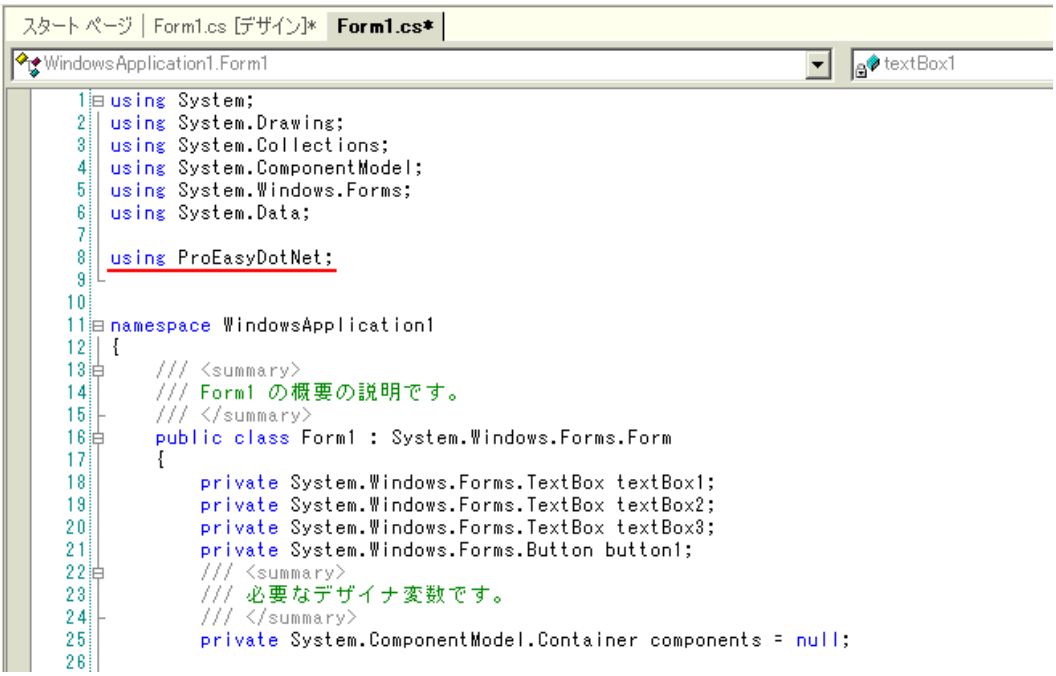
書き込み関数がクリップボードにコピーされます。



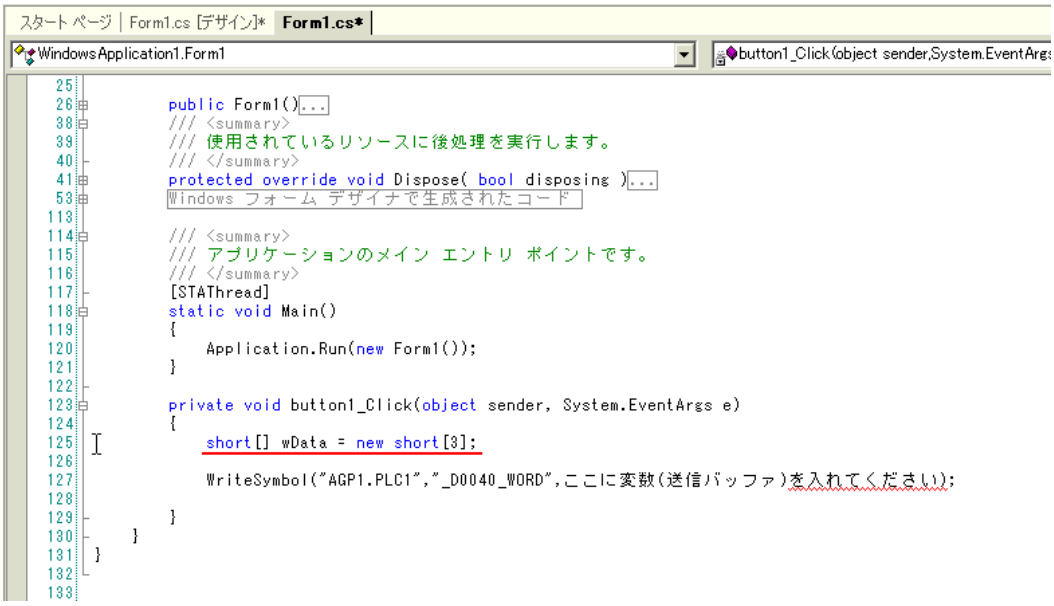
24［Form1］上の［button1］をダブルクリックし、［button1_Click］メソッド（文字列 " private void button1_Click... "）の下にクリップボードの内容（書き込み関数）を貼り付けます。



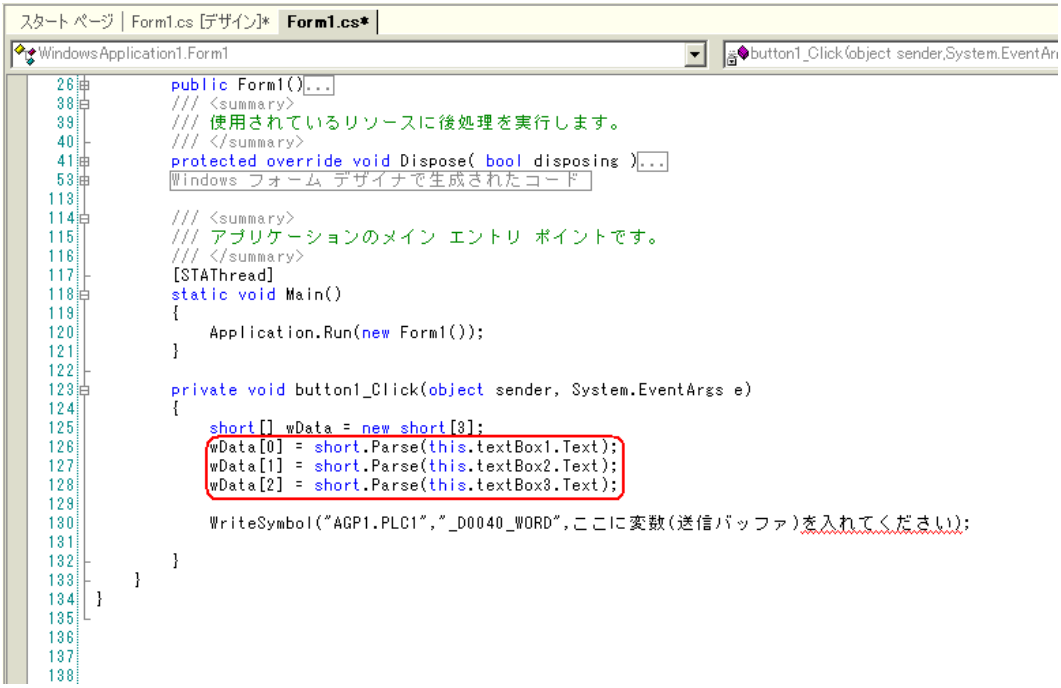
25 ProEasyDotNet のディレクティブを記述します。
ソースコードの先頭に存在する “ using... ” が記述されている部分の一番下に、“ using ProEasyDotNet; ” と入力します。



26 書き込むデータを格納するエリアとして、変数 “ wData ” を宣言します。
配列の型（本例では “ Short ”）は、対象となるシンボルのデータタイプに合わせてください。長さは、対象となるシンボルと同じ長さ（本例では “ 3 ”）を指定します。

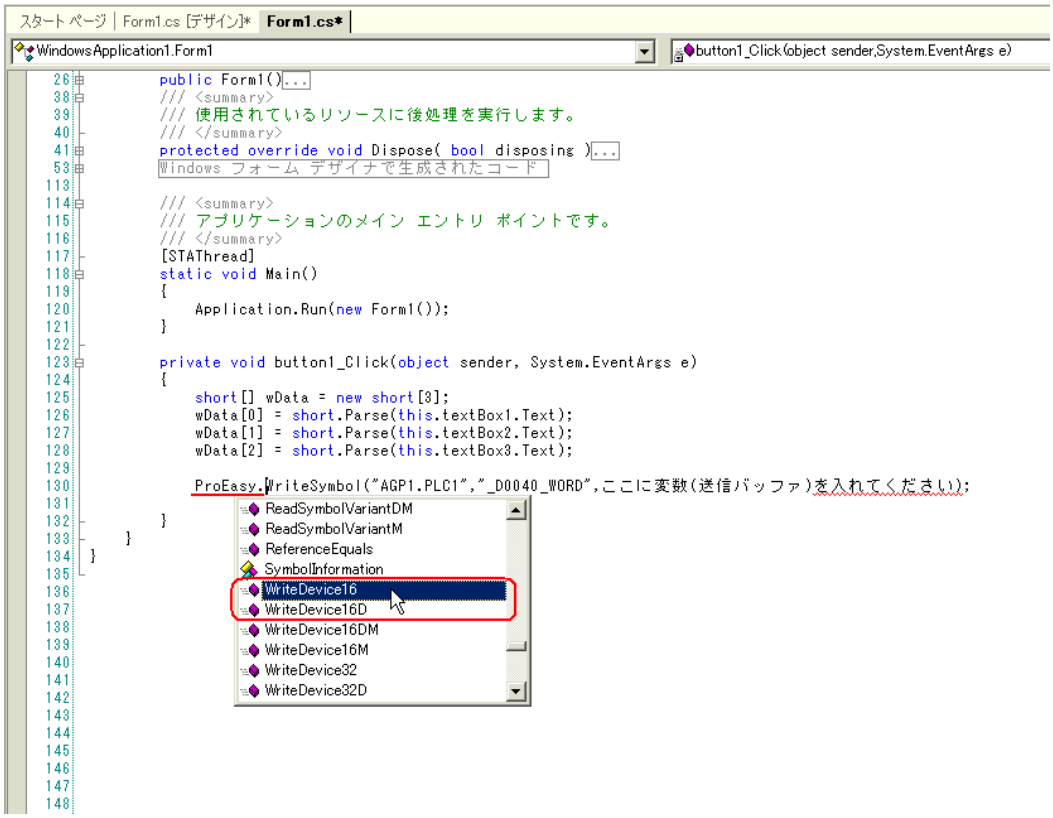


27［textBox1］～［textBox3］に入力するデータを、配列にセットします。

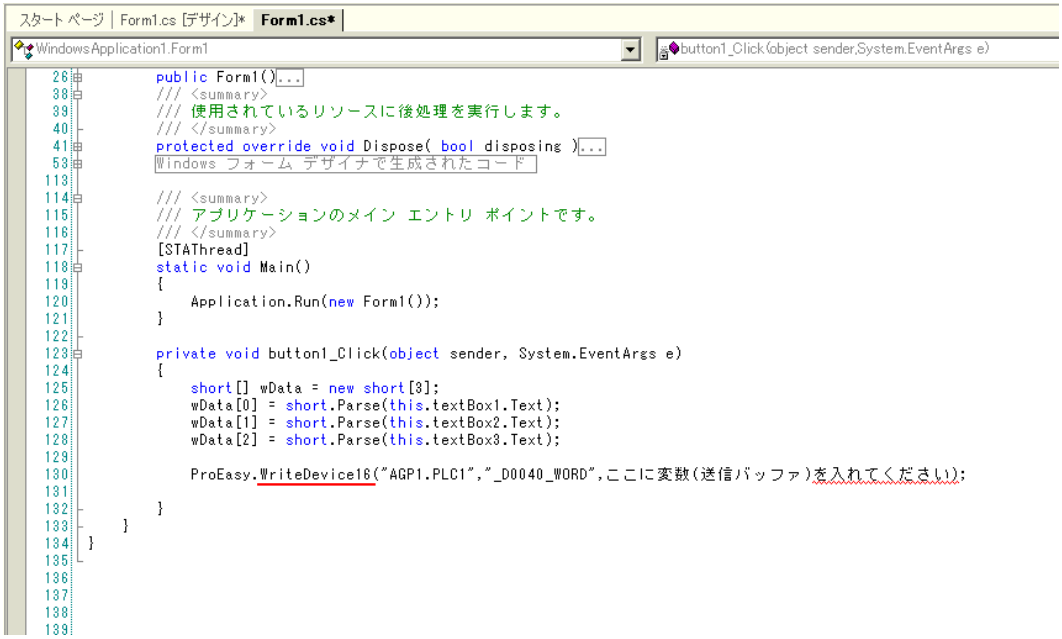


```
スタート ページ | Form1.cs [デザイン]* | Form1.cs*
WindowsApplication1.Form1 | button1_Click(object sender,System.EventAr
 26          public Form1()...
 38          /// <summary>
 39          /// 使用されているリソースに後処理を実行します。
 40          /// </summary>
 41          protected override void Dispose( bool disposing )...
 53          Windows フォーム デザイナで生成されたコード
113
114          /// <summary>
115          /// アプリケーションのメイン エントリ ポイントです。
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",ここに変数(送信バッファ)を入れてください);
131
132          }
133      }
134  }
135
136
137
138
```

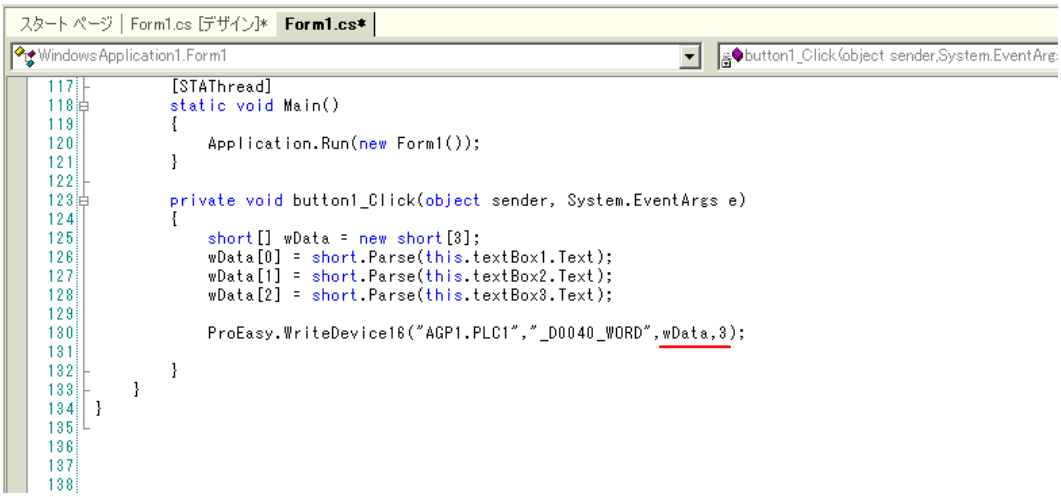
28 “ WriteSymbol ” の前に “ ProEasy. ” と入力し、表示されるリストボックスの中から［WriteDevice16］を選択します。



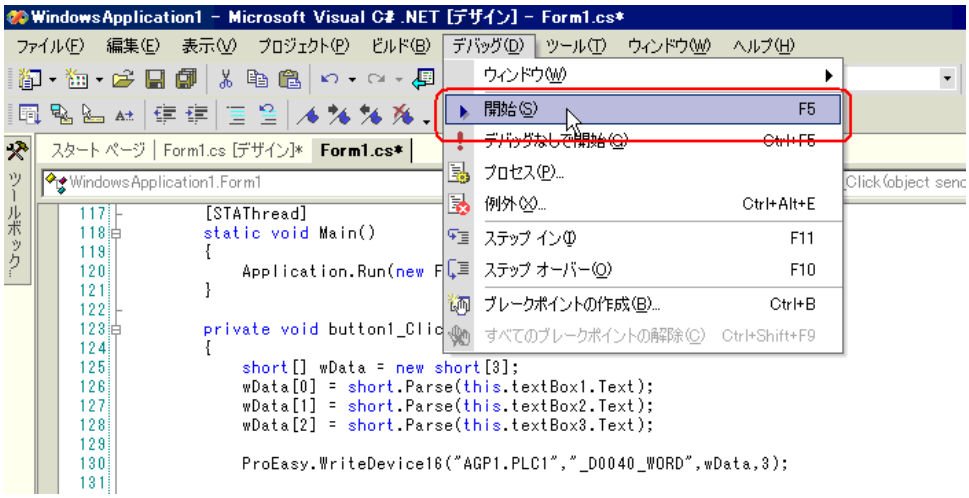
29 クリップボードから貼り付けた文字列（書き込み関数）の “ WriteSymbol ” を削除します。



30 3 番目の引数として、データを格納するエリア “ wData ” を指定します。その後ろに “ , ”（カンマ）を入力し、4 番目の引数として、対象とするシンボルの長さ “ 3 ” を入力します。



31 メニューの［デバッグ］から［開始］を選択します。



32 起動直後には、［TextBox］に文字列 “ textBox* ” が表示されています。

書き込むデータ（3 点分）を［TextBox］に入力したあと、［button1］をクリックすると、データがシンボルで指定した箇所に書き込まれます。