面向对象的方法，属性，动作，接口

1.Object ‘Action’ 目标'动作'

Symbol: 

In an action, you implement additional program code, which you can further implement as in another language as the base implementation. This base implementation is a function block or a program where you inserted the action.

An action does not have its own declaration and it works with the data from the base implementation. This means that the action uses the input and output variables and the local variables from its base implementation.

Add an *Action* to a function block or program by clicking *Project* › *Add Object* › *Action*

在一个操作中，您实现的附加程序代码，可以将在其基础中使用另外另一种语言来进一步实现。在此基础实现是插入操作的函数块或程序。
一个动作是没有自己声明的变量，它与主程序中的数据一起工作。这意味着该操作使用输入和输出变量都是来自其主程序中的变量。
通过单击项目将动作添加到功能块或程序中

Add Action

Name	Name of the action
Implementation language	Drop-down list of implementation language

- Input assistance when creating inheriting blocks
- Calling an action

Input assistance when creating inheriting blocks

When you do object-oriented programming and want to use inheritance for blocks, you have the following support: When you insert a method, action, etc. below an inherited block, the *Add Object* dialog box includes a combo box with a list of methods, actions, etc. used in the base block. In this way, you can easily accept a method definition of the base and adapt it accordingly for the inherited method of the block. Methods and attributes with the `PRIVATE` access modifier are not available in this selection because they should not be inherited. When accepted into the inherited block, methods and attributes with the `PUBLIC` access modifier automatically have a blank access modifier field. (Functionally, this means the same thing.)

当您进行面向对象编程并希望对块使用继承时，您有以下的支持：当在继承的块下插入方法、动作等时，添加对象对话框包括一个包含方法、动作列表的组合框，在此基础块中使用等。这样，您就可以很容易地接受基（主程序）的方法定义，并相应地适应块的继承方法。在此选择中，不可用带有私有访问修饰符的方法和属性，因为它们不应该被继承。当被接受到继承的块中时，具有公共访问修饰符的方法和属性自动具有空白访问修饰符字段。（从功能上讲，这意味着同样的事情。）

See also

- Extension of Function Blocks
- Object-Oriented Programming

相关案例

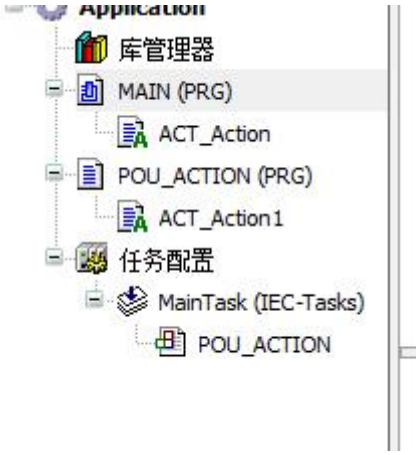
Syntax:

A.定义程序变量

```
PROGRAM MAIN
VAR
    Inst : CTD;       //CTD 的使用,必须先将 load 的值置 0-1-0 的过程，才能启动减计数。

    In: BOOL;
    Erg: BOOL;
    A:INT:=5;
    B:INT:=10;
    C:INT;
END_VAR
```

B.添加动作 ACT_Action

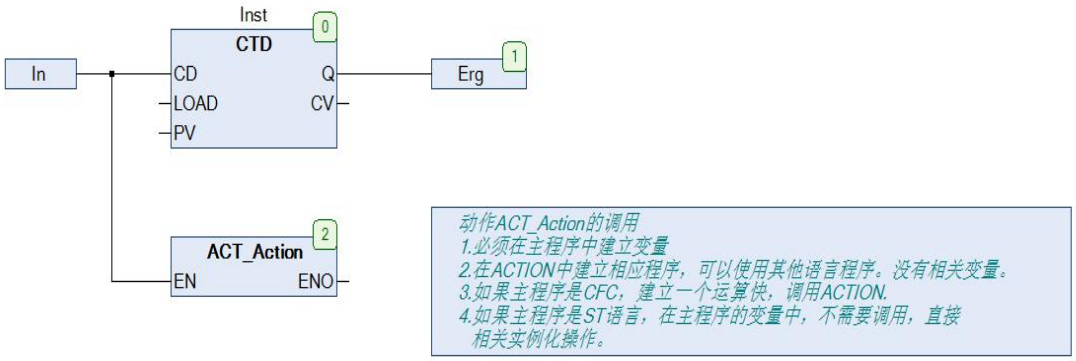


C.在 ACT_Action 编写程序。

IF　IN THEN　　//动作程序中可以使用其他语言，例如 ST 语言。他和主程序共同完成整个控制。

　C:=A+B;　　//动作的理解，在主程序中完成一部分的操作程序，感觉使用这种语言不方便。可以做一个动作

D.在主程序中编写程序



如果主程序中使用 ST 语言调用动作，如下所示：

```
Inst(
    CD:=In ,
    LOAD:= ,
    PV:=50 ,
    Q=>Erg,
    CV=> );

ACT_Action1();              (*动作ACT_Action的调用
                              1.必须在主程序中建立变量
                              2.在ACTION中建立相应程序，可以使用其他语言程序。没有相关变量。
                              3.如果主程序是ST语言，在主程序的变量中，不需要实例化申明，直接
                                调用相关实例化。*)
```

2.Object ‘Method’ '目标'方法'

Symbol: 

Keyword: `METHOD`

Methods are an extension of the IEC 61131-3 standard and a tool for object-oriented programming that is used for data encapsulation. A method contains a declaration and an implementation that includes a series of statements. However, unlike a function, a method is not an independent POU, and it is assigned to a function block or program. You can use interfaces for the organization of methods.

方法是 IEC 61131-3 标准的扩展和用于数据封装的面向对象编程工具。方法包含一系列语句的声明和实现。但是，与函数不同的是，方法不是独立的 pou，它被分配到函数块或程序中。您可以使用接口来访问方法。

You can add a method below a program or a function block. Click *Project* ‣ *Add object* ‣ *Method* to open the *Add method* dialog.

您可以在程序或功能块下添加方法。单击项目“添加对象”方法打开“添加方法”对话框。

- Declaration 声明
- Implementation 执行情况
- Calling a method 调用方法
- Recursive method call 递归方法调用
- Special methods of a function block 函数块的特殊方法
- Dialog *Add method* 对话框添加方法
  Input assistance when creating inheriting blocks 创建继承块时的输入帮助

Declaration

- The variables of a method contain temporary data and are valid only during the execution of the method (stack variables). All variables that are declared and implemented in a method are reinitialized each time the method is called.
- Methods that are declared in an interface may define input, output and `VAR_IN_OUT` variables and may not contain any implementation.
- Like functions, methods can have additional outputs. You must assign these additional outputs in the method call.
- Depending on the declared access modifier, a method can be called only within its own namespace (`INTERNAL`), only within its own programming module and its derivatives (`PROTECTED`), or only within its own programming module (`PRIVATE`). For `PUBLIC`, the method can be called from anywhere.

782/5000

声明

1.方法的变量包含临时数据，且仅在执行方法（堆栈变量）期间有效。每次调用方法时，都会重新初始化方法中声明和实现的所有变量。
2.在接口中声明的方法可以定义输入、输出和输出变量，并且可以不包含任何实现。
3.和函数一样,方法可以有额外的输出。您必须在方法调用中指定这些附加输出。
4.根据声明的访问修饰符，一个方法只能在它自己的命名空间（内部）内调用，只能在它自己的编程模块及其衍生物（受保护）内调用，或者只能在它自己的编程模块（私有）内调用。对于公共，该方法可以从任何地方调用。

Implementation

- Access to function block instances or program variables is permitted in the implementation of the method.
- The `THIS` pointer allows for access to its own function block instance. Therefore, the pointer is permitted only in methods that are assigned to a function block.
- A method cannot access `VAR_TEMP` variables of the function block.
- A method can call itself recursively.

执行情况

1.在方法的实现中允许访问函数块实例或程序变量。
2.此指针允许访问它自己的函数块实例。因此，指针只允许在分配给函数块的方法中使用。
3.方法不能访问函数块的变量。
方法可以递归地调用自己

Hint

If you copy a method below a POU and add it below an interface, or move the method there, then the contained implementation is removed automatically.

Calling a method

Syntax for calls:

```
<return value variable> := <POU name>.<method name>(<method input name> := <variable name> (, <further method input name> := <variable name> )* );
```

For the method call, you assign transfer parameters to the input variables of the method. Pay attention to the declaration when doing this. However, it is enough to specify the names of the input variables without paying attention to their order in the declaration.

对于方法调用，将向方法的输入变量分配传输参数。这样做的时候要注意宣言。但是，只要指定输入变量的名称，而不注意它们在声明中的顺序就足够了。

Example

Declaration

```
METHOD PUBLIC DoIt : BOOL
VAR_INPUT
    iInput_1 : DWORD;
    iInput_2 : DWORD;
    sInput_3 : STRING(12);
END_VAR
```

Call

```
bFinishedMethod :=DoIt(sInput_3 := 'Hello World ', iInput_2 := 16#FFFF, iInput_1 := 16);
```

When the method is called, the return value of the method is assigned, for example, to variables declared locally.

However, you can omit the names of the input variables, in which case you have to pay attention to the declaration order.

Example

Declaration

```
METHOD PUBLIC DoIt : BOOL
VAR_INPUT
    iInput_1 : DWORD;
```

```
    iInput_2 : DWORD;
    sInput_3 : STRING(12);
END_VAR
```

Call

```
bFinishedMethod := fbInstance.DoIt( 16, 16#FFFF,'Hello World ');
```

Recursive method call

Within the implementation, a method can call itself, either directly by means of the `THIS` pointer, or by means of a local variable for the assigned function block.

- `THIS^.<method name>(<parameter transfer of all input and output variables>)` : Direct call of the affected function block instance with the `THIS` pointer.
- `VAR fb_Temp : <function_block_name>; END_VAR` : Call by means of a local variable of the method that instantiates the affected function block temporarily.

A compiler warning is issued for such a recursive call. If the method is provided with the pragma `{attribute 'estimated-stack-usage' := '<sstimated_stack_size_in_bytes>'}`, then the compiler warning is suppressed. For an implementation example, refer to the section “Attribute ‘estimated-stack-usage’”.

在实现中，方法可以直接通过此指针或通过所分配的函数块的局部变量调用自己。

1. 这个^。<方法名称>-lrb<所有输入和输出变量的参数转移>-rb-:使用此指针直接调用受影响的函数块实例。
2. var fb_time:<函数_区块名;endvar:通过临时实例化受影响函数块的方法的局部变量调用。

对这种递归调用发出编译器警告。如果该方法提供了杂注{属性'估计积载使用':='<sstamed_size_n_byte>'}，则编译器警告将被抑制。有关实现示例，请参见“属性‘估计积载使用’”部分。

To call methods recursively, it is not enough to specify only the method name. If only the method name is specified, then a compiler error is issued: *Program name, function, or function block instance expected instead of ...*

See also

- Attribute ‘estimated-stack-usage’
- THIS

Special methods of a function block

See also

`FB_Init`	Declarations are automatically implicit. Explicit declaration is also possible. Contains initialization code for the function block, as is defined in the declaration part of the function block.
`FB_Reinit`	Explicit declaration is necessary. After the instance of the function block is copied (as during an online change): calls and reinitializes the new instance module.
`FB_Exit`	Explicit declaration is necessary. Call for each instance of the function block before a new download or a reset or during an online change for all shifted or deleted instances.
Attributes and interface attribute	Provides `Set` and/or `Get` accessor methods.

- Methods 'FB_Init', 'FB_Reinit', and 'FB_Exit'
- Object 'Property'
- Object 'Interface property'

Dialog *Add method*

Function: Configures the method that is added by closing the dialog.

Call: *Project* ‣ *Add object* ‣ *Method* ; context menu

Name	Example: `meth_DoIt`
Return type	Default data type or structured data type of return value Example: `BOOL`
Implementation language	Example: *Structured Text (ST)*
Access specifier	Controls access to data. •

	PUBLIC or not specified: Access is not restricted. • • *PRIVATE*: Access is restricted to the program, function block, or GVL. • The object is marked as `(private)` in the POU or device view. The declaration contains the keyword `PRIVATE`. • • *PROTECTED*: Access is restricted to the program, function block, or GVL with its derivations. The declaration contains the keyword `PROTECTED`. • The object is marked as `(protected)` in the POU or device view. • • *INTERNAL*: Access to the method is restricted to the namespace (library). • The object is marked as `(internal)` in the POU or device view. The declarat ion contains the keyword `INTERNAL`. •
Add	Adds a new method below the selected object.

Requirement: A program (`PRG`) or a function block (`FUNCTION_BLOCK`) is selected in the *POUs* view or the *Devices* view.

Input assistance when creating inheriting blocks

When you do object-oriented programming and want to use inheritance for blocks, you have the following support: When you insert a method, action, etc. below an inherited block, the *Add Object* dialog box includes a combo box with a list of methods, actions, etc. used in the base block. In this way, you can easily accept a method definition of the base and adapt it accordingly for the inherited method of the block. Methods and attributes with the `PRIVATE` access modifier are not available in this selection because they should not be inherited. When accepted into the inherited block, methods and attributes with the `PUBLIC` access modifier automatically have a blank access modifier field. (Functionally, this means the same thing.)

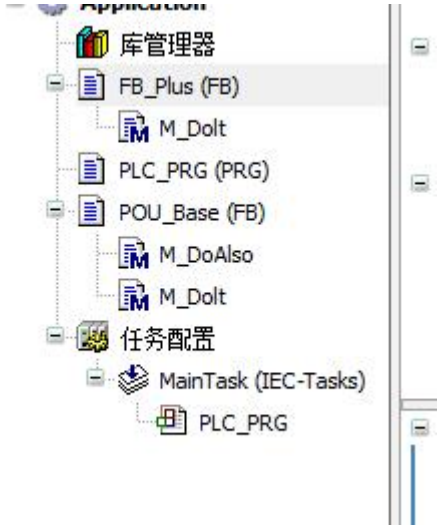
当您进行面向对象编程并希望对块使用继承时，您有以下的支持：当在继承的块下插入方法、动作等时，添加对象对话框包括一个包含方法、动作列表的组合框，在基块中使用的等。这样，您就可以很容易地接受基的方法定义，并相应地适应块的继承方法。在此选择中，不可用带有私有访问修饰符的方法和属性，因为它们不应该被继承。当被接受到继承的块中时，具有公共访问修饰符的方法和属性自动具有空白访问修饰符字段。（从功能上讲，这意味着同样的事情。）

相关案例

此案例着重讲解方法的继承作用。

A. 新建主程序，新建父类功能块以及其功能块的变量和方法自身量。

新建子类功能块以及其功能块的变量和方法自身的变量



1. 父类的变量及其程序

```
FUNCTION_BLOCK POU_Base

VAR_INPUT
```

```
END_VAR
VAR_OUTPUT
END_VAR
VAR
    iCntBase:INT;
    iMessageBase:STRING;
END_VAR
```

2. 父类方法 M_Dolt 的变量及其程序

```
METHOD M_Dolt : BOOL
VAR_INPUT
END_VAR
```

```
iCntBase:=-1;
iMessageBase:='SUPER';
```

3. 父类方法 M_DoAlso 的变量及其程序

```
METHOD M_DoAlso : BOOL
VAR_INPUT
END_VAR
```

```
M_DoAlso:=TRUE;
```

B. 新建子类的功能块，并且通过 Extend 的继承，指向父类继承关系

1.编写功能块的相关变量

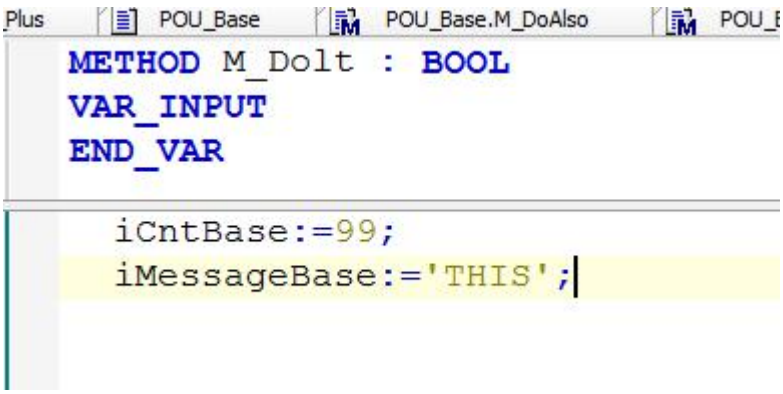
```
FUNCTION_BLOCK FB_Plus EXTENDS POU_Base
VAR_INPUT
    Blocal_1:BOOL;
    Blocal_2:BOOL;
END_VAR
```

```
VAR_OUTPUT
    icnt:INT;
    sMessage:STRING;
END_VAR
VAR
    Rusult:bool;
END_VAR
```

2.编写程序

```
IF Blocal_1 THEN
    THIS^.M_Dolt();                          //调用子类的方法
    icnt:=THIS^.iCntBase;                    //调用子类的方法中数据
    sMessage:=THIS^.iMessageBase;            //读取子类的方法中字符串
ELSIF Blocal_2 AND NOT Blocal_1 THEN
    SUPER^.M_DoAlso();                       //调用父类的方法
    Rusult:=TRUE;
ELSE
    SUPER^.M_Dolt();                         //调用父类的方法
    icnt:=SUPER^.iCntBase;                   //从写父类的方法中数据
    sMessage:=SUPER^.iMessageBase;           //从写父类的方法中数据
END_IF
```

3.编写子类 M_Dolt 功能块中的方法程序



3.Object ‘Interface’目标'接口'

Symbol: 

Keyword: INTERFACE

An interface is a means of object-oriented programming. The object ITF describes a set of method and property prototypes. In this context, prototype means that the methods and properties contain only declarations and no implementation.

This allows different function blocks having common properties to be used in the same way. An object *ITF* is added to the application or the project with the command *Project* ‣ *Add object* ‣ *Interface* .

Adding an interface

Inheritance	
Name	Interface name
Extends	: Extends the interface that you enter in the input field or via the input assistant . This means that all methods of the interface that extend the new interface are also available in the new interface.

You can add the objects *Interface property* and *Interface method* to the object *ITF*. Interface methods may contain only the declarations of input, output and input/output variables, but no implementation.

So that you can also use an interface in the program, there must be a function block that implements this interface.

This means:

- the function block contains the interface in its IMPLEMENTS list in its declaration part
- the function block contains an implementation for all methods and property prototypes of the interface

A function block can implement one or more interfaces. You can use the same method with identical parameters, but different implementation code in different function blocks.

Please note the following:

- You may not define variables within an interface. An interface has no implementation part and no actions. Only a collection of methods is defined, in which you may define only input, output and input/output variables.
- CODESYS always treats variables declared with the type of an interface as references.
- A function block that implements an interface must contain implementation code for the methods of the interface. You have named the methods exactly as in the interface and the methods contain the same input, output and input/output variables as in the interface.

Hint

Interface references and online change The following can happen with a compiler version < 3.4.1.0: if a function block changes its data because variables are added or deleted, or because the type of variables changes, then CODESYS copies all instances of the function block to a new memory location. In this case, however, an interface reference refers not to the new memory location, but still to the old one.

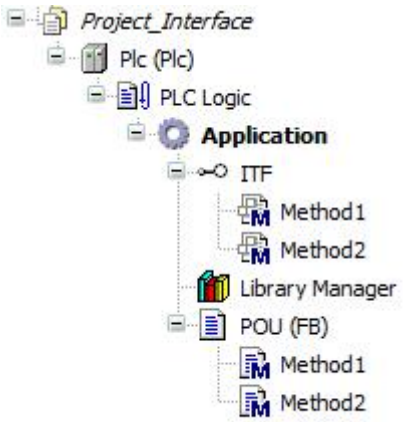
In case of compiler versions >= 3.4.1.0, CODESYS automatically re-addresses the interface references so that CODESYS also references the correct interface in case of an online change. CODESYS requires additional code and more time for this, so that jitter problems can occur depending on the number of objects concerned. Therefore, CODESYS displays the number of variables and interface references concerned before the execution of the online change and you can then decide whether the online change should be executed or aborted.

Example

Definition of an interface and its use in a function block

You have inserted the interface *ITF* below the application. The interface contains the methods *Method1* and *Method2*. *ITF*, *Method1* and *Method2* contain no implementation code. You insert the required variable declarations only in the declaration part of the methods.

If you subsequently insert a function block in the device tree that implements the interface *ITF*, CODESYS automatically also inserts the methods *Method1* and *Method2* under the function block. Here you can implement function-block-specific code in the methods.



Implementing Interfaces

Implementing interfaces is based on the concept of object-oriented programming. With common interfaces, you can use different but similar function blocks the same way.

A function block that implements an interface has to include all methods and attributes that are defined in that interface (interface methods and interface attributes). This means that the name and the inputs and outputs of the methods or attributes must be exactly the same. When you create a new function block that implements an interface, CODESYS adds all methods and attributes of the interface automatically to the tree below the new function block.

Hint

If you add more interface methods afterwards, then CODESYS does not add these methods automatically to the affected function block. To perform this update, you must execute the *Implement Interfaces* command explicitly.

For inherited function blocks, you have to make sure that any methods or attributes that were derived through the inheritance of an interface also receive the appropriate implementation. Otherwise they should be deleted in case the implementation that was provided in the basis should be used. Respective compile error messages or warnings are displayed, prompted automatically by added pragma attributes. For more information, refer to the help page for the *Implementing Interfaces* command.

Hint

- You must assign the interface of a function block to a variable of the interface type before a method can be called via the variable.
- A variable of the interface type is always a reference of the assigned function block instance.

A variable of the interface type is a reference to instances of function blocks. This kind of variable can refer to every function block that implements the interface. If there is no assignment to a variable, then the variable in online mode contains the value `0`.

Examples

The `I1` interface contains the `GetName` method.

```
METHOD GetName : STRING
```

The functions blocks `A` and `B` implements the interface `I1`:

```
FUNCTION_BLOCK A IMPLEMENTS I1
FUNCTION_BLOCK B IMPLEMENTS I1
```

For this reason, both function blocks must include a method named `GetName` and the return type `STRING`. Otherwise the compiler reports an error.

A function includes the declaration of a variable of interface `I1` type.

```
FUNCTION DeliverName : STRING
VAR_INPUT
    l_i : I1;
END_VAR
```

Function blocks that implement the `I1` interface can be assigned to these input variables.

Examples of function calls:

```
DeliverName(l_i := A_instance); // call with instance of type A
DeliverName(l_i := B_instance); // call with instance of type B
```

Calling of interface methods:

In this case, it depends on the actual type of `l_i` whether the application calls `A.GetName` or `B.GetName`.

```
DeliverName := l_i.GetName();
```

- Command ‘Implement interfaces’

- Implementing an interface in a new function block
- Implementing an interface in an existing function block

Implementing an interface in a new function block

1.Right-click *Application* in the device tree and select *Project* ‣ *Add Object* ‣ *POU* .

⇒ The *Add POU* dialog box opens.

2.Type the name for the new POU in the *Name* input field, for example *POU_Im*.

Select *Function block*.

3.Click *Implemented* and then the more button ().

4.In the input assistant, select the interface from the category *Interfaces*, for example ITF1, and click on *OK*.

5.To insert more interfaces, click  and select a another interface.

6.As an option, you can select an *Access modifier* for the new function block from the selection list.

7.Select from the *Implementation language* combo box (example: *Structured text (ST)*.

Click *Add*.

⇒ CODESYS adds the *POU_Ex* function block to the device tree and opens the editor. The first line contains the text:

```
FUNCTION_BLOCK POU_Im IMPLEMENTS ITF1
```

1. The interface and its methods and properties are
2. now inserted below the function block in the
3. device tree. Now you can type program code into
4. the implementation part of the interface and its methods.

Implementing an interface in an existing function block

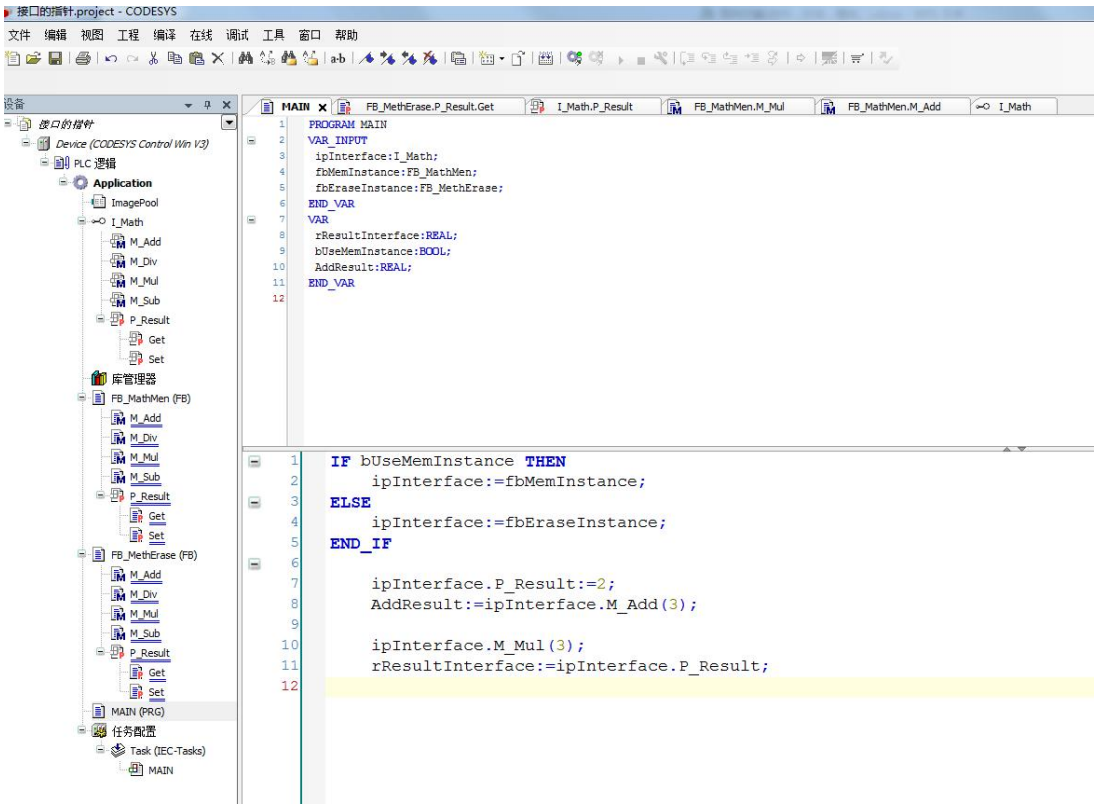
1.Double-click the *POU_Ex(FB)* POU in the device tree.

⇒ The POU editor opens.

Extend the existing entry in the uppermost line `FUNCTION_BLOCK POU_Im` with `IMPLEMENTS ITF1`

⇒ The *POU_Im* function block implements the *ITF1* interface

相关案例



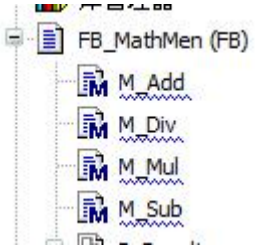
B. 新建接口，定义接口中的方法。

```
METHOD M_Add : REAL
VAR_INPUT
    rVal:REAL;
END_VAR
```

c. 定义接口中的属性。

C. PROPERTY P_Result : REAL

D. 定义功能块，并且指向建好的接口。



E. 在功能块的方法中，定义算法，属性。

```
{warning '添加方法实现'}
METHOD M_Add : REAL
VAR_INPUT
    rVal    : REAL;
END_VAR
```

```
Rmen:=Rmen+Rval;
M_Add:=Rmen;
```

F. 在主程序中调用接口，调用功能块。

```
PROGRAM MAIN
VAR_INPUT
    ipInterface:I_Math;                 //接口的实例化
    fbMemInstance:FB_MathMen;           //其中一个功能块的实例化
    fbEraseInstance:FB_MethErase;       //另一个功能块的实例化
END_VAR
VAR
    rResultInterface:REAL;              //接收数据变量的定义
```

```
bUseMemInstance:BOOL;          //接收条件变量的定义
AddResult:REAL;
END_VAR
```

G.　编写主程序。

```
IF    bUseMemInstance    THEN
ipInterface:=fbMemInstance;                    //接口指向功能块 1
ELSE
ipInterface:=fbEraseInstance;                  //接口指向功能块 2
END_IF
    ipInterface.P_Result:=2;                   //给接口属性赋值
AddResult:=ipInterface.M_Add(3);               //输出调用接口方法的值
ipInterface.M_Mul(3);                          //输出调用接口方法的值
rResultInterface:=ipInterface.P_Result;        //读取接口属性的值
```

4.Object ‘Property’目标'属性'

Symbol: 

Keyword: `PROPERTY`

Properties are an extension of the IEC 61131-3 standard and a tool for object-oriented programming. Properties are used for data encapsulation because they allow for external access to data and act as filters at the same time. For this purpose, a property provides the accessor methods *Get* and *Set* which allows for read and write access to the data of the instance below the property.

You can add a property with accessor methods below a program, a function block, or a global variable list. Click *Project* ‣ *Add object* ‣ *Property* to open the *Add property* dialog.

关键字：属性

属性是 IEC 61131-3 标准的扩展和面向对象编程的工具。属性用于数据封装，因为它们允许外部访问数据并同时充当筛选器。为此目的，属性提供了访问

器方法的获取和设置，允许对属性下面的实例的数据进行读写访问。
可以在程序、函数块或全局变量列表下添加具有访问器方法的属性。单击“项目”、“添加对象”属性以打开“添加属性”对话框。

Note

You can added an interface property below an interface.

If you copy a property that is below a POU and add it below an interface, or if you move the property there, then the included implementations are removed automatically.

See also

注：
您可以在接口下添加接口属性。
如果您复制了 pou 下面的属性并将其添加到接口下面，或者如果您将该属性移动到那里，则会自动删除包含的实现。
另见

- Object ‘Interface property’

- Dialog *Add property*
- Editor ‘Property’
- Get and Set accessors 获取并设置访问器
- Monitoring of properties in online mode 以在线模式监察物体属性
- Input assistance when creating inheriting blocks 创建继承块时的输入帮助

Dialog *Add property*

Function: Configures the property that is added by closing the dialog.

Call: *Project* › *Add object* › *Property* ; context menu

Requirement: A program (`PRG`), a function block (`FUNCTION_BLOCK`), or a global variable list (`GVL`) is selected in the *POUs* view or *Devices* view.

Name	Name (identifier) of the property
Return type	Default type or structured type of return value Example: `BOOL`
Implementation language	Example: *Structured Text (ST)*
Access specifier	Controls access to data. •

	PUBLIC or not specified: Access is not restricted. • • *PRIVATE*: Access is restricted to the program, function block, or GVL. • The object is marked as `(private)` in the POU or device view. The declaration contain s the keyword `PRIVATE`. • • *PROTECTED*: Access is restricted to the program, function block, or GVL with its derivations. • The object is marked as `(protected)` in the POU or device view. The declaration contains the keyword `PROTECTED`. • • *INTERNAL*: Access is restricted to the namespace (library). • The object is marked as `(internal)` in the POU or device view. The declaration contains the keyword `INTERNAL`. •
Add	Adds a new property be low the selected object and below that the accessor methods *Get* and *Set*.

	Note: When you select a property, you can also add a previously removed accessor explicitly by clicking *Add object*.

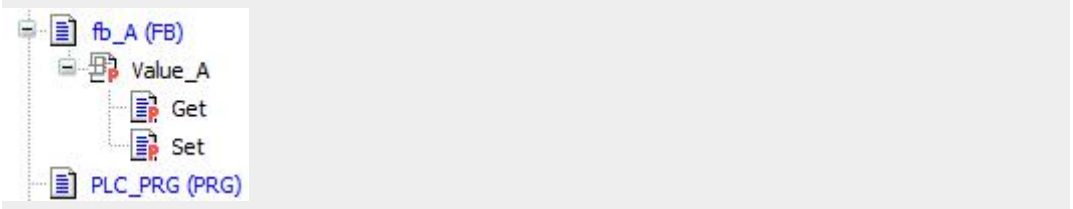
Editor ‘Property’

You can program the data access in the editor. The code can contain additional local variables. However, it cannot contain any additional input variables or (as opposed to a function or method) output variables.

您可以在编辑器中对数据访问进行编程。代码可以包含额外的本地变量。但是，它不能包含任何额外的输入变量或（相对于函数或方法）输出变量

Example

相关案例



1.功能块变量的定义

```
FUNCTION_BLOCK fb_A                                    //定义功能块 FB_A

VAR_INPUT

END_VAR

VAR_OUTPUT

END_VAR

VAR

    iA : INT;                                          //定义变量 IA

    iB : INT;                                          //定义变量 IB

END_VAR
```

2.功能块程序

```
iA := iA + 1;
```

3.功能块中名称为 Value_A 的属性定义

```
Property Value_A
```

```
PROPERTY PUBLIC Value_A : INT
```

4.通过属性 Value_A，来读取变量 IA 的数值。

```
fb_A.Value_A.Get
```

```
VAR
END_VAR
Value_A := iA;
```

5.通过属性 Value_A，将数据写入变量 IB 中。

```
fb_A.Value_A.Set
```

```
VAR
END_VAR
iB := Value_A;
```

6.通过属性，来对封装功能块中的数据，进行访问或者写入数据。

Get and Set accessors

When the *Set* accessor is called, the property is written. It is then used like an input parameter. When the Get accessor is called, the property is read. It is used like an output parameter. Access is restricted via access modifiers, and the objects are marked accordingly.

Dialog *Add Get (Set) accessor*

Implementation language	Example: *Structured Text (ST)*
Access specifier	Access modifiers

	• `PUBLIC` or not specified: Access is not restricted. • • `PRIVATE`: Access is restricted to the program, function block, or GVL. • The object is marked as `(private)` in the POU or device view. The declaration contains the keyword. • • `PROTECTED`: Access to the property is restricted to the program, function block, or GVL and its derivations. The declaration contains the keyword. • The object is marked as `(protected)` in the POU or device view. • • `INTERNAL`: Access to the method is restricted to the namespace (library). • The object is marked as `(internal)` in the POU or device view. The declaration contains the keyword. •
Add	Adds the accessor methods *Get* or *Set* below the selected property.

When a property is accessed as read only or write only, you can delete the unneeded accessors.

You can add accessors explicitly by selecting a property and clicking *Add object*. A dialog opens, either *Add Get accessor* or *Add Set accessor*. There you can set the implementation language and the access.

Monitoring of properties in online mode

以在线模式监察物体属性

The following pragmas are provided for the monitoring of properties in online mode. You insert them at the top position of the property definition:

- `{attribute 'monitoring:= 'variable'}`: Each time the property is accessed, CODESYS saves the momentary value in a variable and displays the value of this variable. This value can become outdated if no further access to the property takes place in the code.
- `{attribute 'monitoring' := 'call'}`: **Each time** the value is displayed, CODESYS calls the code of the `Get` accessor. If this code contains a side effect, then the monitoring executes the side effect.

以下备注用于在线模式下的属性监控。将它们插入到属性定义的顶部位置：
1.{属性 监视：=变量}:每次访问属性时，代码将瞬间值保存在变量中，并显示此变量的值。如果在代码中没有对属性的进一步访问，此值可能变得过时。
2.{属性 监视：=调用;}：每次显示值时，codesy 会调用 get 访问器的代码。如果此代码包含一个副作用，则监视执行该副作用

You can monitor a property with the help of the following functions.

Inline monitoring

Requirement: The option *Activate inline monitoring* is selected in the category *Text editor* of the *Options* dialog.

Watch list

See also

- Calling Methods 调用“方法”
- Attribute ‘monitoring’ 属性“监视”

Input assistance when creating inheriting blocks

When you do object-oriented programming and want to use inheritance for blocks, you have the following support: When you insert a method, action, etc. below an inherited block, the *Add Object* dialog box includes a combo box with a list of methods, actions, etc. used in the base block. In this way, you can easily accept a method definition of the base and adapt it accordingly for the inherited method of the block. Methods and attributes with the `PRIVATE` access modifier are not available in this selection because they should not be inherited. When accepted into the inherited block, methods and attributes with

the `PUBLIC` access modifier automatically have a blank access modifier field. (Functionally, this means the same thing.)

See also

创建继承块时的输入帮助

当您进行面向对象编程并希望对块使用继承时，您有以下的支持：当在继承的块下插入方法、动作等时，添加对象对话框包括一个包含方法、动作列表的组合框，在基块中使用的等。这样，您就可以很容易地接受基的方法定义，并相应地适应块的继承方法。在此选择中，不可用带有私有访问修饰符的方法和属性，因为它们不应该被继承。当被接受到继承的块中时，具有公共访问修饰符的方法和属性自动具有空白访问修饰符字段。（从功能上讲，这意味着同样的事情。）
另见

5.Object ‘Transition’ 目标“过渡”

Symbol: 

This object can be used as a transition element in a program block implemented in SFC.

See also

此对象可作为在 SFC 中实现的程序块中的过渡元素。
另见

- SFC Elements ‘Step’ and ‘Transition’

Note

When you do object-oriented programming and want to use inheritance for blocks, you have the following support: When you insert a method, action, etc. below an inherited block, the *Add Object* dialog box includes a combo box with a list of methods, actions, etc. used in the base block. In this way, you can easily accept a method definition of the base and adapt it accordingly for the inherited method of the block. Methods and attributes with

the PRIVATE access modifier are not available in this selection because they should not be inherited. When accepted into the inherited block, methods and attributes with the PUBLIC access modifier automatically have a blank access modifier field. (Functionally, this means the same thing.)

注：
当您进行面向对象编程并希望对块使用继承时，您有以下的支持：当在继承的块下插入方法、动作等时，添加对象对话框包括一个包含方法、动作列表的组合框，在基块中使用的等。这样，您就可以很容易地接受基的方法定义，并相应地适应块的继承方法。在此选择中，不可用带有私有访问修饰符的方法和属性，因为它们不应该被继承。当被接受到继承的块中时，具有公共访问修饰符的方法和属性自动具有空白访问修饰符字段。（从功能上讲，这意味着同样的事情。）

See also 另见

- Extending Function Blocks
- Programming Applications