APPLICATION NOTE

AC500 V3 OOP KEYWORDS

DESCRIPTIONS AND EXAMPLES

Contents

1 Introduction

1.1 Scope of the document

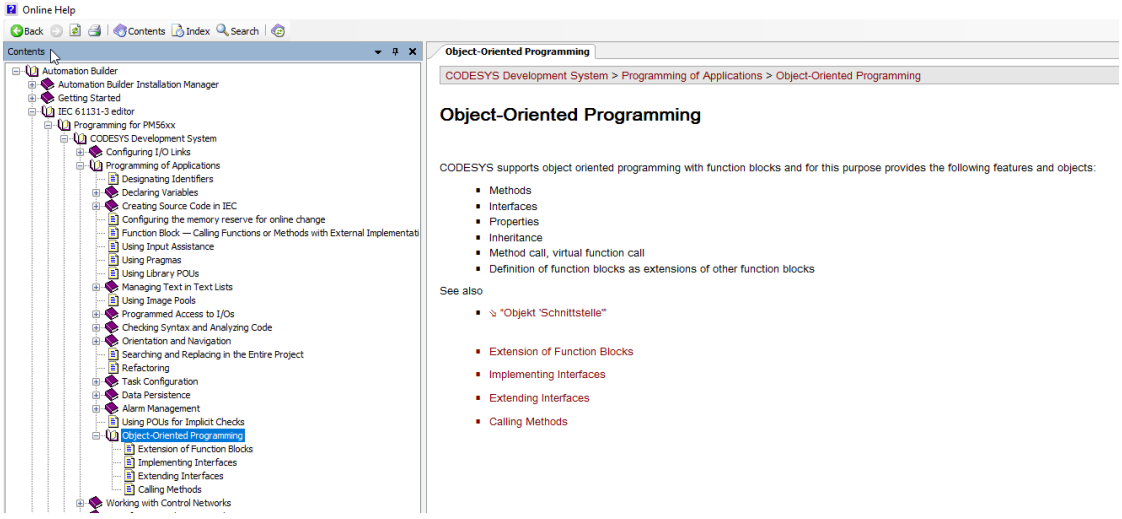
This document explains the new keywords in CoDeSys V3 that can be used for Object Oriented Programming (OOP), improving programming style, debugging, … Although there are separate capters for each keyword, it is recommended that you also read the previous as they might explain basic principles which are assumed later.

1.2 Compatibility

The application note explains the functionalities and Keywords that can be used with an AC500 V3. Some features are linked to several Firmware versions. More information can be found in the online help.

1.3 Overview

The main topic of this document is the Object Oriented Programming. This document is not describing a real use case but more the possibilities which can be archived with OOP. Detailed descriptions can be found in the Automation Builder Online help.



As an addition to the Automation Builder online help three whitepapers from Plan Automation-Technology are used.[1]

- GOTO OOP
- Benefiting from OOP in IEC61131-3
- OOP in IEC61131-3 for experts

This document can give an overview how OOP can be realized in an AC500. Furthermore, it explains some basics about OOP. In addition, other Keywords are described that can be used to improve the programming style and the robustness.

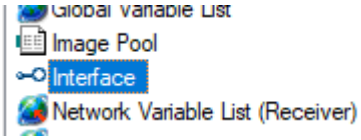
[1] Accessed on 2020-05-19

2 Objects that can be added

This chapter focuses on the objects that can be added to an existing application, interface or function block. Only a few objects are described in this note. In the Automation Builder online help all objects can be found. This chapter is just an introduction and does not contain any information about OOP or other keywords.

2.1 Interfaces

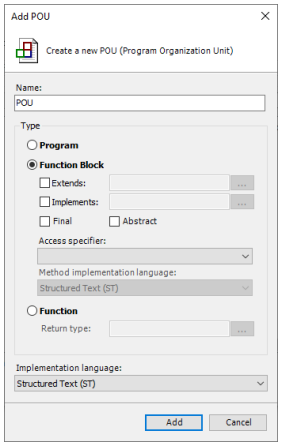
An interface can be added to an application.



An interface can itself have method and property prototypes. This means an interface contains only declarations but no implementation. This allows different function blocks to have the identical interface. This allows to use different function blocks in the same way. For more information please refer to chapter 3.1.2.

2.2 Function Blocks

When adding an POU to the application Function Block can be selected.



The Function Block is the main Object for OOP and can be treated as class which can have several instances. The new possibilities in V3 which extend the functionality in V2 are described in chapter 3.1.

2.3 Action

An action can be added to any POU. The action is additional program code which doesn't have any own variables. The action is using the resources from its base implementation.

The Action can be programmed in another language as the Base fb. In addition the action can be also called from other POUs by calling <fb_instance>.<action>;

2.4 Method

A Method is an extension for OOP for data encapsulation similar to Actions. The main difference is that Methods have own Input Variables, Local Variables and a Return. Furthermore, a Method can also use the variables which are used in the base fb similar to the Action.

Similar to a function a method can return a value at the end of the call. It is not possible to wait for a done flag or have an internal step chain inside a method. All local variables will be initialized with the next program call. In contrast to functions Methods are linked to a function block or program.

Methods can also be called from other POU without calling the Base function block: <fb_instance>.<method> or <POU>.<method>;

2.5 Property

A Property is an extension for OOP. It can be added to a POU or a global variable list. The behavior of a Property is similar to a variable. A Property can be read and written. In case the property is read the get function is called. Here the definition of reading a variable can be realized. By writing the set function is called.

The read and write via a Property has two advantages. The direct reading/writing of fb variables can be controlled. A set method can prohibit a write to the variables. In addition, a scaling can be realized. Furthermore, two intern variables of the fb can be shown as one to the instantiating POU. For example, a tank control function block has the property Tank_FillingLevel. By reading the property the variable currentFillLevel is returned. By writing Tank_FillingLevel the variable desiredFillLevel is changed.

2.6 Transition

Transitions are usually used for SFC programming. A transition is a condition which can be True or False. In SFC the Transition is needed if multiple instructions are used. The transitions are not highlighted in this document.

3 Keywords

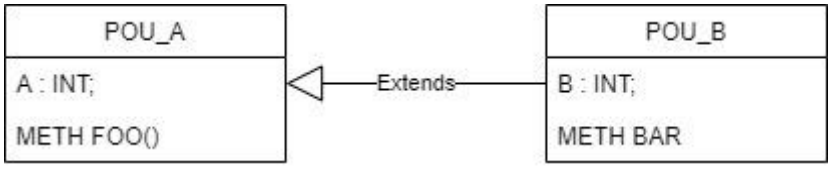
This chapter gives an overview about new Keywords which can be user to program an AC500 V3.

3.1 Inheritance

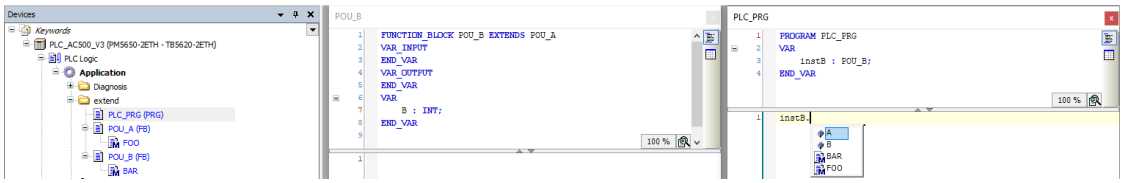
Inheritance is the key feature of object oriented programming (OOP).

3.1.1 Extends

One Function Block can extend another one. Details to function blocks can be found in chapter 2.2.



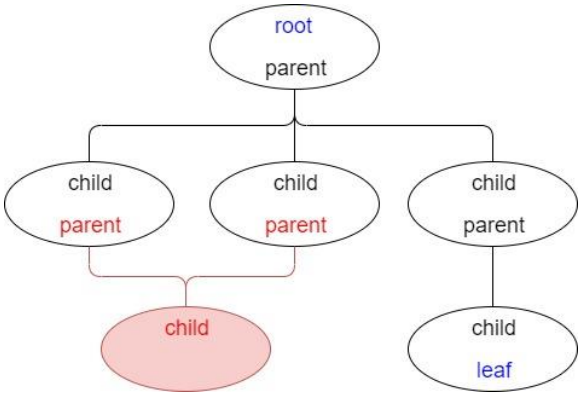
As shown above the POU_B extends the POU_A. Below a screenshot from the same implementation in Automation Builder is visible. By adding the POU_B the extension of POU_A can be selected. On the left side the two function blocks with the method FOO or BAR are visible. In the middle the implementation of POU_B is shown where the Extend of POU_A can be seen in the first line. The local variable B is declared below. On the right side POU_B is instantiated in PLC_PRG as instB. As visible this instance has not only the variable B and the Method BAR as members but also the method BAR and the Variable A from POU_A. No reimplementation in the extending function block is necessary.



As POU_B is extending POU_A it inherits all Attributes, Properties and Methods of POU_A.

Everything which is described here for the inheritance between two function blocks is the same for interfaces. One interface can also extend another one and inherit the methods and properties.

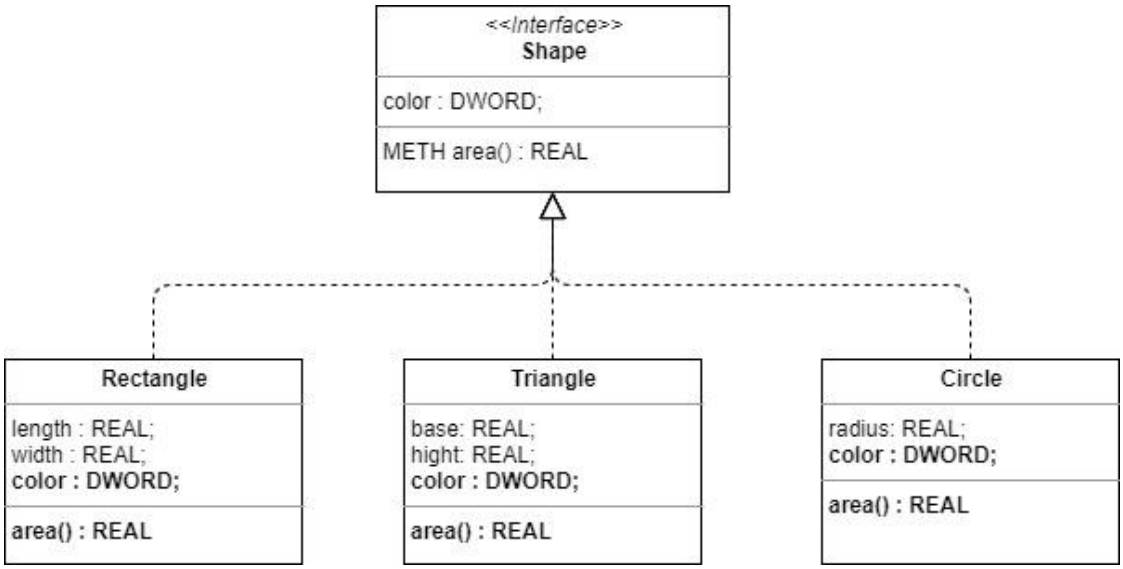
In OOP this is called parent and child class. The child will inherit all properties and methods of his parent. A parent which has no parent itself is called root. A child which doesn't have any children itself is called leaf. Each Parent can have several children. But a child has only one parent class. The described tree is shown in the picture below. In blue the root and the leaf are highlighted. In red a not allowed class is shown. Because two parents are not allowed.



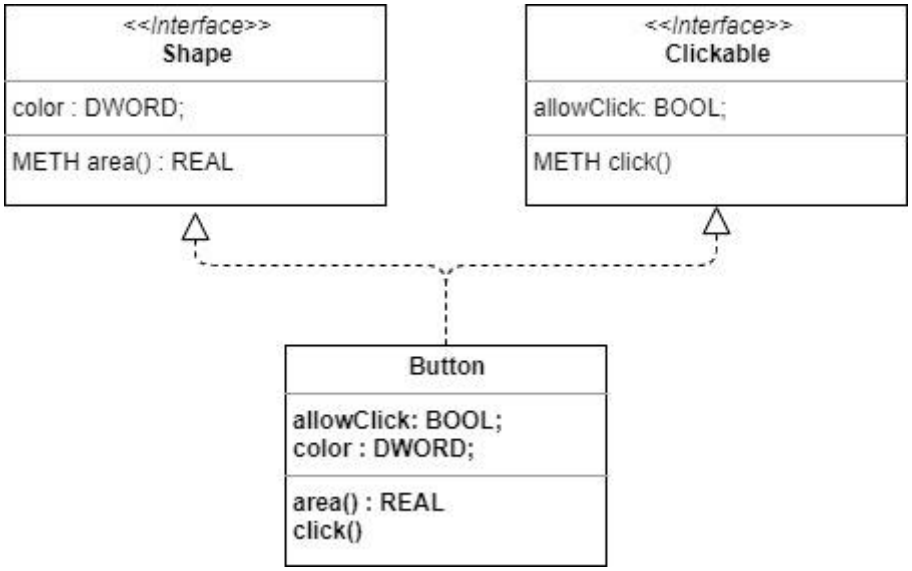
3.1.2 Implements

Similar to the function block which can extend another function block an Interface can be implemented by a function block. The function block which implements an interface inherits all methods and properties from the interface. As the interface is only a prototype the implementation has to be done in the function block. Detailed information about Interfaces can be found in chapter 2.1.

The advantage of interfaces is, that all function blocks which implement one interface behave the same way. As visible in the picture below the interface Shape can be implemented by different classes. As shape has the method area. All classes that implement Shape have also to implement the method area. Even if the implementation of this method is different in Rectangle, Triangle and Circle a POU which instantiates these function blocks can call the Method area in the same way. Even an exchange of the function block would be possible as the interface is still the same.

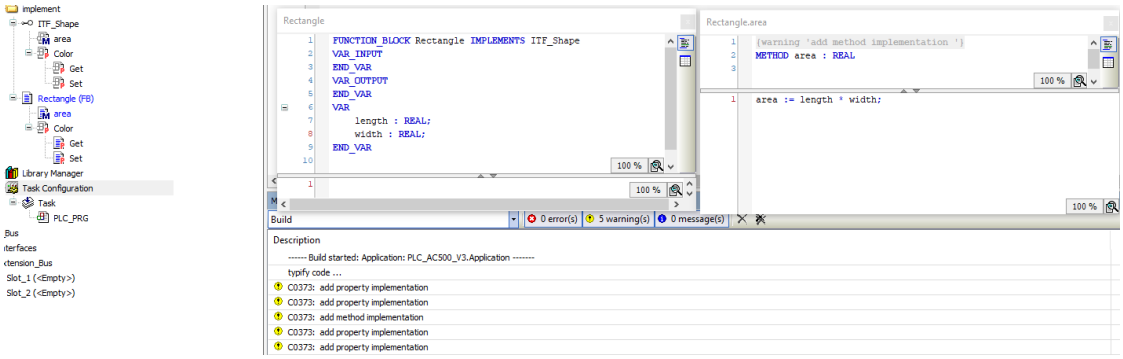


In contrast to the extending of function blocks one function block can implement multiple interfaces. For example, the function block Button can implement the interface Clickable and Shape.

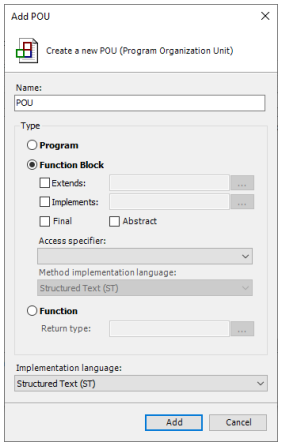


Also a combination of Implementing and Extending is possible. Similar pictures shown above a Button could also implement Clickable and extend Rectangle. Then the method area is predefined in Rectangle.

When implementing an interface in Automation Builder the prototype Methods and Properties are added automatically to the function block. As visible in the picture below in the middle Rectangle Implements ITF_Shape. On the left the method area as well as the Color Property is visible below Rectangle. By compiling some warnings are thrown. On the right the Method Rectangle.area() is visible. In the first row the warning is thrown to remind the programmer to add an implementation and delete this warning afterwards.



3.1.3 Final



When adding a function block Final can be checked in addition to the extends and implements. This means that the function block is a leave and cannot be extended by another function block. When trying to extend a function block with the attribute Final the compiler throws an error.

3.1.4 Abstract

Similar to Final also Abstract can be checked when adding a function block. An abstract function block is a prototype which cannot be instantiated directly. When trying to instantiate a function block with the attribute Abstract the compiler throws an error.

Similar to an interface an abstract class cannot be instantiated directly. Another function block is needed. Following table shows the differences between them.[2]

Interface	Abstract class
A fb can implement multiple interfaces	A fb can extend only one Abstract class
Can have abstract Methods and Properties	Can have abstract or concrete Methods and Properties
By changing the interface all implementing fbs have to change the implementation of this method / property	By changing the abstract class all extending fbs. Inherit this changes by default.
Interfaces does not have access modifiers.	Abstract classes can have access modifiers
Interfaces cannot have local variables	Abstract classes can have local variables

3.1.5 This & Super

Each function block has a THIS pointer available. This is a Pointer to its own instance. In contrast to the THIS Pointer SUPER is a Pointer to the Parent function block.

As usage for the This pointer shadowing is used as example. A function block has the local variables A and B which are assigned to 1 and 2.

PLC_AC500_V3.Application.PLC_PRG.Myfb

Expression	Type	Value
A	INT	1
B	INT	2

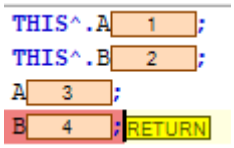
The function block itself has a method which also has the variables A and B that are 3 and 4.

PLC_AC500_V3.Application.PLC_PRG.Myfb.Meth

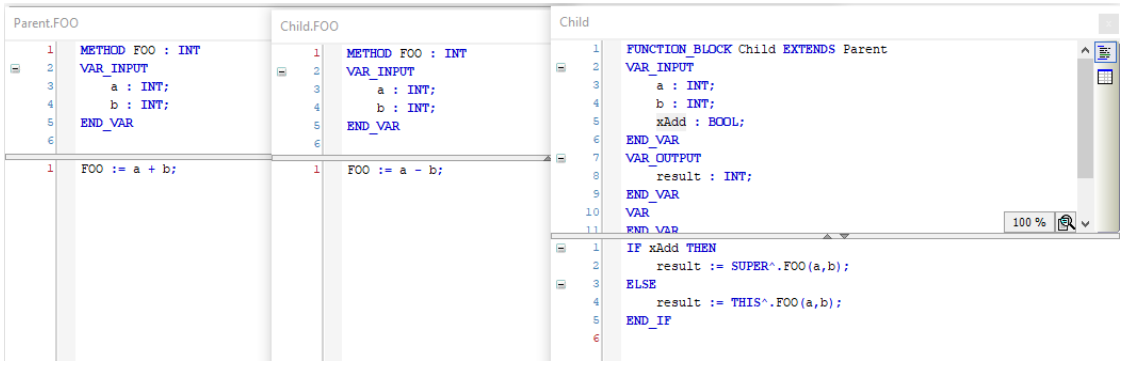
Expression	Type	Value
A	INT	3
B	INT	4

[2] Cf. https://www.guru99.com/interface-vs-abstract-class-java.html Accessed 20.05.2020

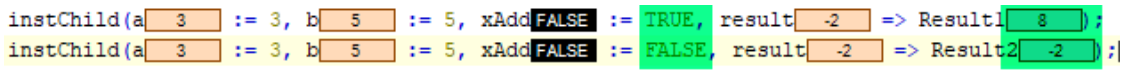
By reading the variable A inside the Method the local variable is read as the variable A from the function block is shadowed by the new variable in the Method. To access also the variables from the function block which are shadowed the This pointer has to be used. That is visible on the screenshot below.



The same shadowing can also be used for Methods itself. The screenshot below shows a implementation in Automation Builder. The Parent function block has the Method FOO where two integer values are added. The Children has also the Method Foo. But here the two values are subtracted. Depending of the input variable xAdd in the Child function block either the own Method (THIS) or the parent Method (Super is called).

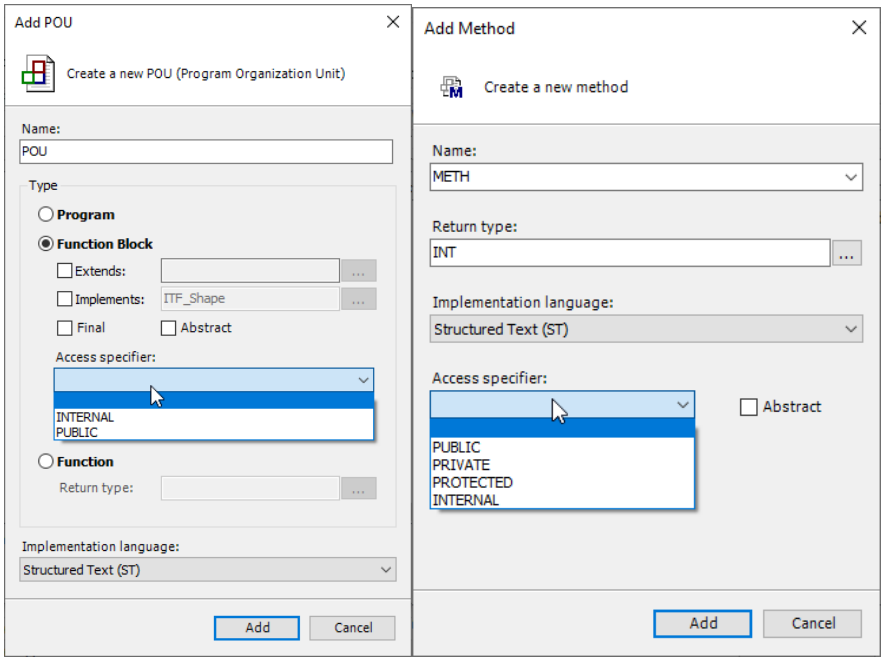


When instantiating the child fb and running it the result will be the Sum when xAdd is TRUE and the Difference when xAdd is FALSE.



3.2 Access Specifier

When adding a function block, method or property an Access specifier can by specified.



This can be INTERNAL and PUBLIC for function blocks and PUBLIC, PRIVATE, PROTECTED and INTERNAL for Methods and Properties.

3.2.1 Public

Public is selected the default setting when no Access is specified. Public means the element can be accessed from an extending function block and any instance.

3.2.2 Protected

Protected means the element can be accessed from an extending function block but not in any instance.

3.2.3 Internal

Internal means the element cannot be accessed from an extending function block but in any instance.

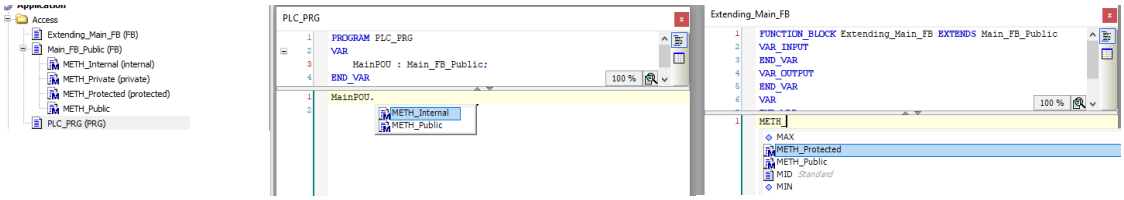
3.2.4 Private

Private means the element can neither be accessed from an extending function block nor in any instance.

3.2.5 Overview

The table and the picture below are showing how methods with different access specifiers can be accessed by extending or instantiating.

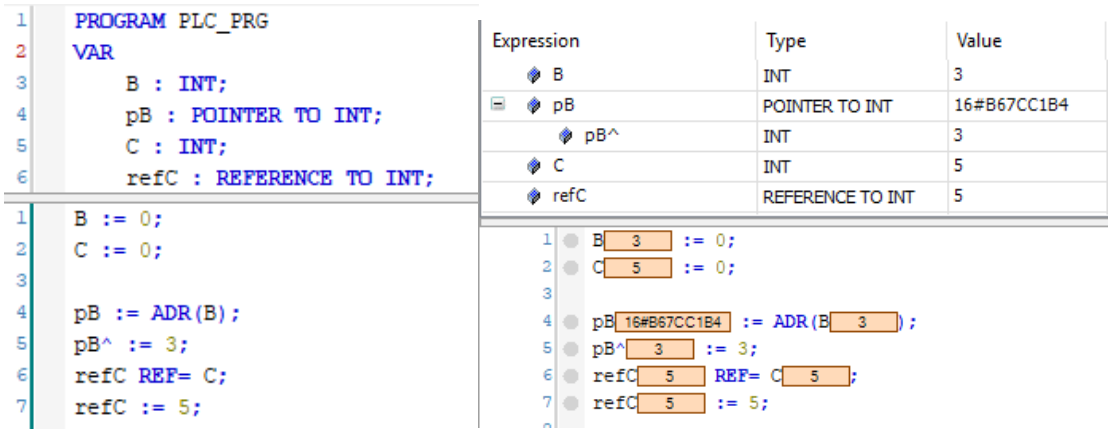
Access specifier	Fb instance	Extending fb
Public	✔	✔
Protected	✘	✔
Internal	✔	✘
Private	✘	✘



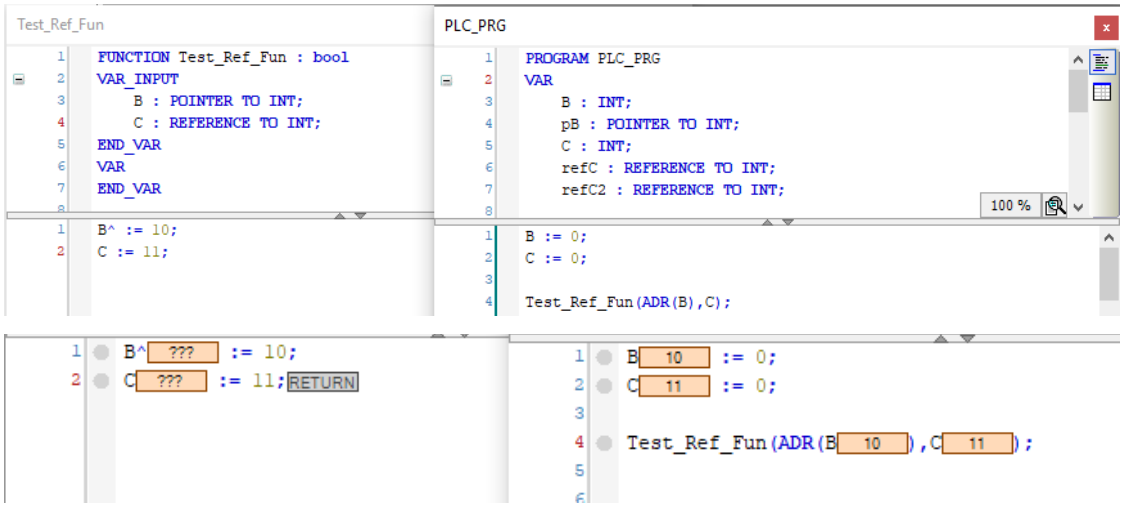
3.3 Reference instead Pointer

Pointers are usually used for arrays or structs to avoid copying many data. A Reference is also a pointer, but it has some advantages.

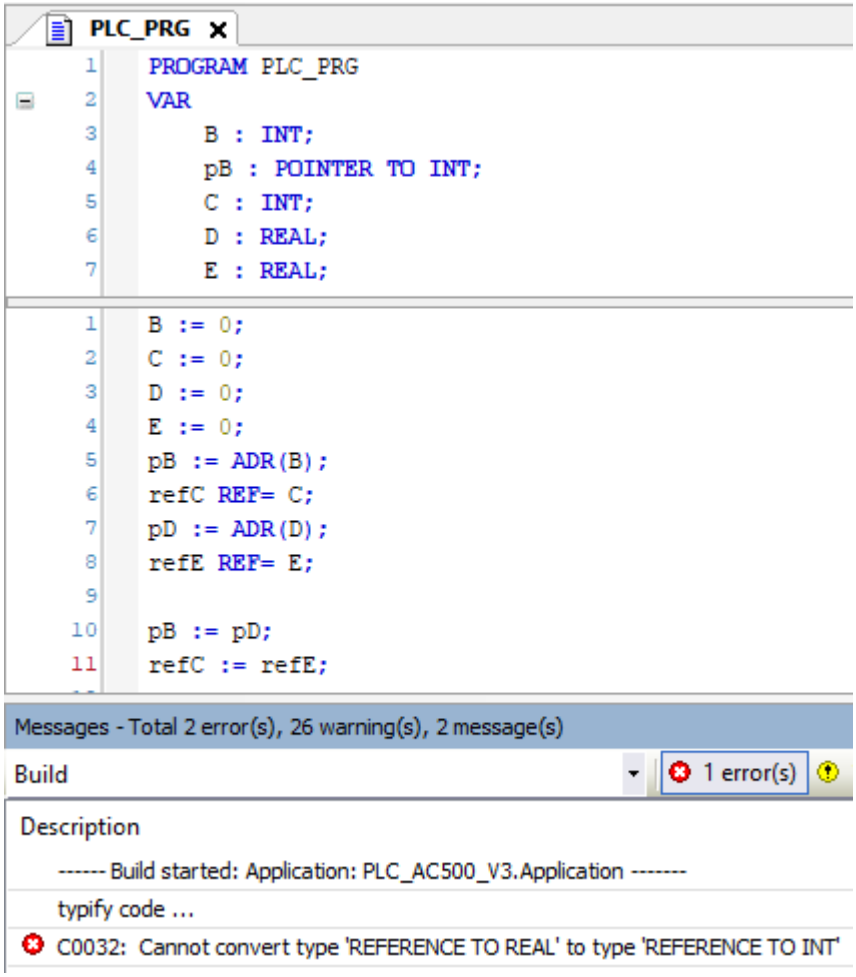
To compare the usage of References with pointers two values B and C are used. B as Pointer and C as Reference. As visible from the screenshot below pB and refC are the pointer/reference to the variables. Assigning a value to a reference variable can just be done without dereferencing the value. This can be seen in line 7.



Furthermore functions with a Reference input can just be assigned to the value direct. No ADR operation is necessary. This can be found in the example below in line 4.



The third advantage is the possibility to check references during compile. This is not possible for pointers. As visible in the screenshot below B and C are integers, D and E are real variables. In line 10 a REAL POINTER is assigned to an INT POINTER. This gives no compile error. Assigning a REAL REFERENCE to an INT REFERENCE like in line 11 is throwing a compiler Error.

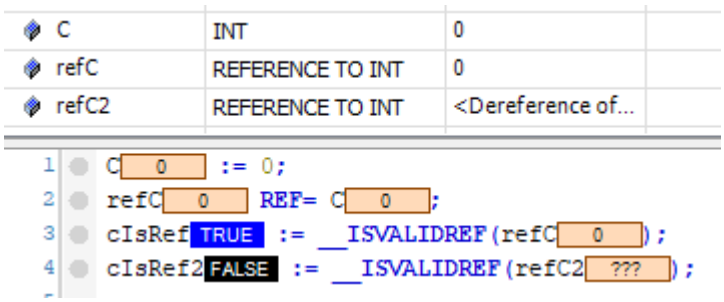


Working with References instead of Pointers is much easier for programming and code readability and furthermore safer to avoid invalid assignment and access.

3.4 Operators

3.4.1 __ISVALIDREF

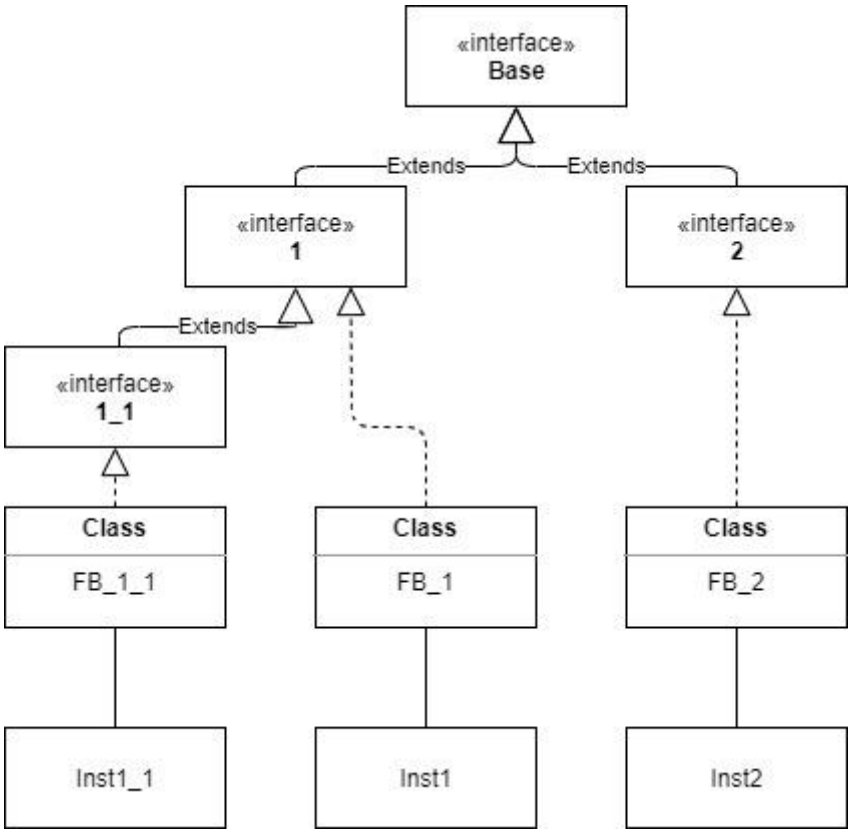
The operator __ISVALIDREF can be used to check if a reference is valid to avoid an invalid access. The usage can be seen in the screenshot below. refC is referencing C but refC2 is not referencing a variable.



3.4.2 __QUERYINTERFACE

QueryInterface is an operator which should be only used by experienced users. The operator can be used for a type conversation of an interface into another. The requirement for the explicit conversation is that both interfaces extend __System.IQueryInterface.

The operator itself is defined as `__QUERYINTERFACE(<ITF_Source>,<ITF_Dest>);` and returns true if the interface conversation was successful. As an example, following layout is used. A Base interface is extended by two other interfaces. One interface has also another child. The tree resulting interfaces have all one implementing function block. In the Program each function block has one instance.



As Inst1, Inst2 and Inst1_1 are all implementing Base following declaration is allowed.

iBase : ITF_BASE := Inst1;

Instead of Inst1 also the other instances can be used to define iBase.

With

```
iBase : ITF_BASE := Inst1;
i1 : ITF_1 := 0;
```

```
__QUERYINTERFACE(Inst1, i1);
```

Inst1 which is decelerated as ITF_BAASE can be assigned to i1 which is ITF_1

Following table shows the result of trying to convert the instances from ITF_BASE to other interfaces.

✔ means the result is TRUE, ✘ means the result is false

Src/Dest	ITF_BASE	ITF_1	ITF_1_1	ITF2
Inst1	✔	✔	✘	✘
Inst1_1	✔	✔	✔	✘
Inst2	✔	✘	✘	✔

A use case for this function is a function which has a reference to an interface as input. Depending on the function block which is inputted to this function different actions can take place.

Similar to this operator __QUERYPOINTER converts an interface to a Pointer.

3.4.3 __TRY, __CATCH, __FINALLY and __ENDTRY

These statements are used for exception handling in the IEC Code. If not allowed statements like a division by 0 are executed an exception is thrown and the PLC goes to stop.

If a statement in TRY throws an exception the PLC executes CATCH instead of going to stop to handle the exception. The statements in FINALLY are called independent if there was an exception or not. These statements are usually used to clean up a program or function block in depended of the success of the execution.

To give a short example where different exceptions can happen a division function is used.

Inputted are A and B as well as a Pointer to the Variable C.

The main logic is in the try part. The C pointer is dereferenced and assigned to the quotient of A and B.

```
FUNCTION Exception_Check : STRING
VAR_INPUT
    A : INT := 0;
    B : INT := 0;
    C : POINTER TO INT;
END_VAR
VAR
    exc : __SYSTEM.ExceptionCode;
END_VAR
```

```
__TRY
    //Try to divide and assign
    C^ := A/B;
__CATCH(exc)
    iNrFailed := iNrFailed +1;
    CASE exc OF
        __SYSTEM.ExceptionCode.RTSEXCPT_ACCESS_VIOLATION:
            Exception_Check := 'RTSEXCPT_ACCESS_VIOLATION';
        __SYSTEM.ExceptionCode.RTSEXCPT_DIVIDEBYZERO:
            Exception_Check := 'RTSEXCPT_DIVIDEBYZERO';
        ELSE
            Exception_Check := DWORD_TO_STRING(exc);
    END_CASE
__FINALLY
    iNrExecuted := iNrExecuted + 1;
__ENDTRY
```

An exception can either be thrown if the pointer to C is invalid and cannot be dereferenced or B is 0 as a division is not possible. In this case the execution of the try part is stopped and catch is called. Depending on the thrown exception a different system behavior is possible. Here the exception name is returned as a string. Each time the Catch part is called iNrFailed is incremented. Independent if the try was successful or the catch has been called finally will be executed. Here iNrExecuted is incremented by 1.

In the main program the function is called

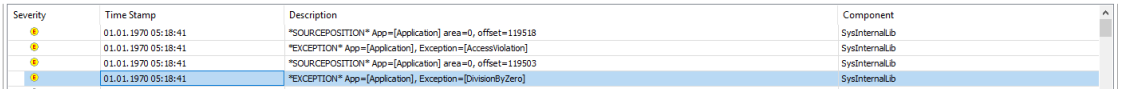
testResult	ARRAY [0..2] OF ST...	
testResult[0]	STRING	''
testResult[1]	STRING	'RTSEXCPT_DIVIDEBYZERO'
testResult[2]	STRING	'RTSEXCPT_ACCESS_VIOLATION'

```
1  CASE test 3 OF
2  0:
3      iNrExecuted 3 := 0;
4      iNrFailed 2 := 0;
5      //This should work
6      A 44 := 44; B 12 := 12;
7      testResult[0] '' := Exception_Check(A 44,B 12,ADR(C 3));
8      test 3 := 1;
9  1:
10     //Dividing by 0 not allowed
11     B 12 := 0;
12     testResult[1] 'RTSEXCPT_D' :=    Exception_Check(A 44,B 12,ADR(C 3));
13     test 3 := 2;
14 2:
15     //0 Pointer cannot be dereferenced
16     B 12 := 12;
17     testResult[2] 'RTSEXCPT_A' :=    Exception_Check(A 44,B 12,0);
18     test 3 := 3;
19
20 END_CASE RETURN
```

In line 6 A and B are set to 44 and 12. The function calculates the integer division 44/12 = 3 successfully. In line 11 B is set to 0. The program tries to divide 44/0 which causes an exception. As visible in the variable testResult at position 1 the reason for this exception was a division by zero. In line 16 B is set to 12 again. In line 17 is visible that the pointer to C is not ADR(C) anymore but 0. The function tries to dereference a zero pointer which causes an exception. As visible in the variable testResult at position 2 the reason for this exception was an access violation.

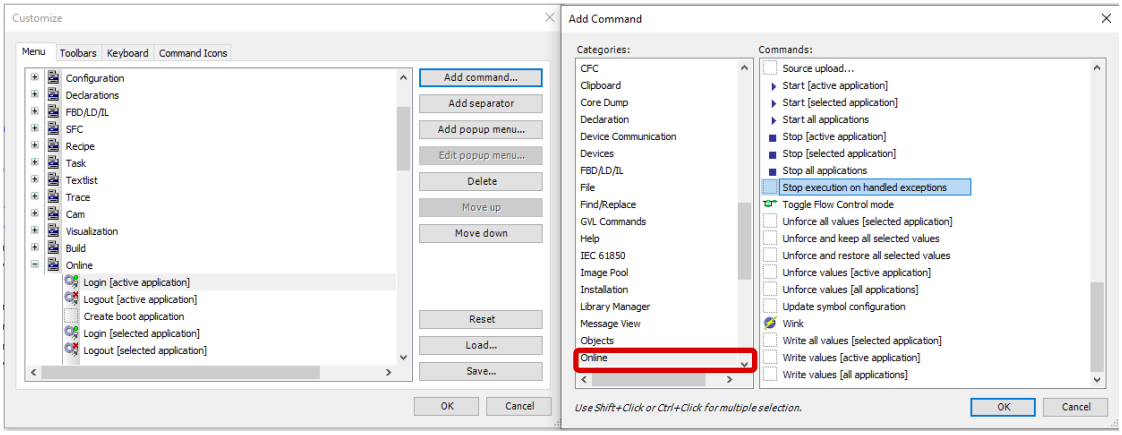
In line 3 and 4 the counter variables are set to 0. As iNrExecuted is 3 now the finally statement was called three times. Accordingly catch was called two times. Depending on the exceptions the Program is still in run and has not stopped.

In the PLC Log the exceptions can be found for debugging.

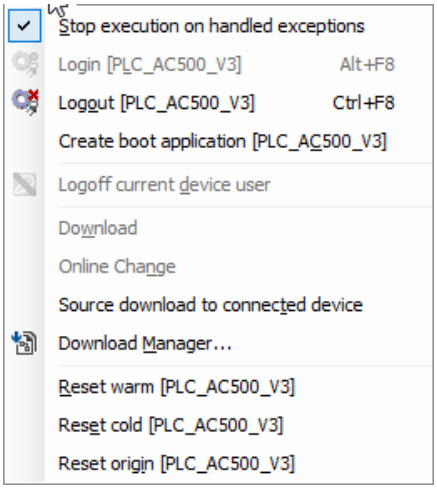


During the development process it might be also necessary to find out state of different variables when the exception happens. Therefore, stop execution on handled exceptions can be selected. This Command has to be added to Automation Builder.

Close all opened projects → Select Tools Customize… → In the Menu tab Online any position can be selected. Click on Add command… → Select “Stop execution on handled exceptions” in the Category “Online”.



When being logged into the PLC the command can be checked.



By running the same Program again, it will stop now. The PLC Log can be used to jump to the source position. Now also the values that cause the exception are visible and the programmer can try to check why B is 0 in this case.

```
1  __TRY
2      //Try to divide and assign
3      C^ 3  := A 44 /B 0 ;
```

3.4.4 __VARINFO

The operator __VARINFO returns a structure containing more information about the variable. The information can be stored in a structure with the type __SYSTEM.VAR_INFO.

As an example, an Array with 10 bytes is used. The screenshot below shows the structure which is be filled by calling infoArr := __VARINFO(bArrVar);

bArrVar	ARRAY [0..9] OF BYTE				This is an array
infoArr	__SYSTEM.VAR_INFO				
ByteAddress	DWORD	3061361184			Address of variable: for bits, address of byte ...
ByteOffset	DINT	10784			Offset in Bytes.
Area	INT	0			Number of Area.
BitNr	INT	255			number of Bit in Bytes...r non-Bit Types this ...
BitSize	UDINT	80			size of variable in Bits
BitAdress	UDINT	0			bitaddress of variable (if variable is at %M/1/...
TypeClass	TYPE_CLASS	TYPE_ARRAY			type class of variable
TypeName	STRING(79)	'ARRAY [0..9] OF BYTE'			type name (for userdeftypes : name of functi...
NumElements	UDINT	10			for arrays : number of base elements
BaseTypeClass	TYPE_CLASS	TYPE_BYTE			for arrays : type class of base type
ElemBitSize	UDINT	8			for arrays : bit size of base element
MemoryArea	MEMORY_AREA	MEM_GLOBAL			area information : me..., input, output, ret...
Symbol	STRING(39)	'bArrVar'			symbol name : input o...erator as string: up ...
Comment	STRING(79)	'This is an array'			comment of variable

ABB Automation Products GmbH
Eppelheimer Straße 82
69123 Heidelberg, Germany
Phone: +49 62 21 701 1444
Fax: +49 62 21 701 1382
E-Mail: plc.support@de.abb.com
www.abb.com/plc