APPLICATION NOTE

AC500 V3 USING PRAGMAS

DESCRIPTIONS AND EXAMPLES

Contents

1 Introduction

1.1 Scope of the document

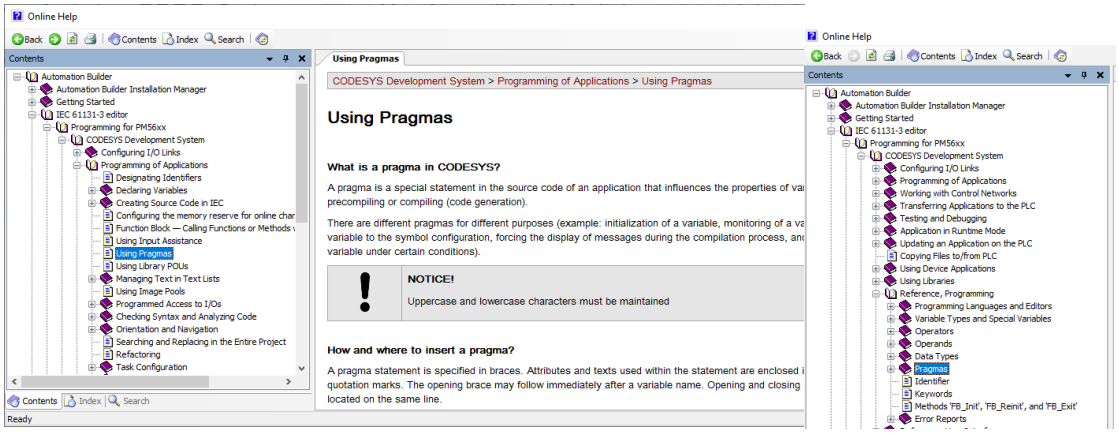
This document explains the functionality of pragmas and possible use cases

1.2 Compatibility

The application note explains some pragmas 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

Pragmas are special statements that influence the behavior when precompiling or compiling (build) the project. This document explains the basic functionalities and use cases. All pragmas with detailed descriptions can be found in the online help.



2 Pragmas

Pragmas are instruction which affect the (pre-) compilation process of one or more variables. Pragmas can be grouped in four categories

1. Message pragmas
2. Region pragmas
3. Conditional Pragmas
4. Attribute Pragmas

Pragmas are written in between curly brackets.

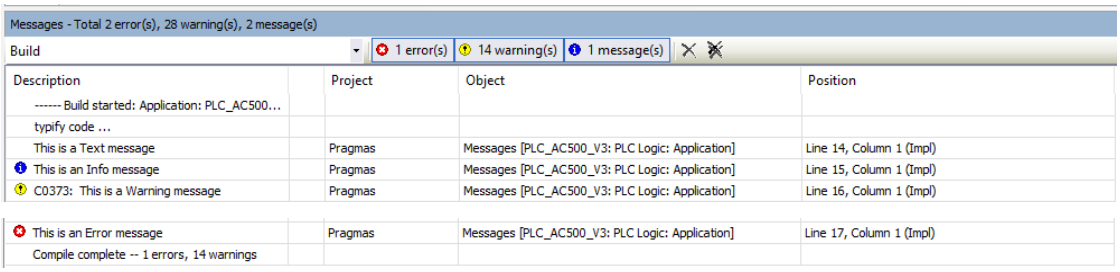
2.1 Message pragmas

Message pragmas output a message during compilation process.

Four Message types are possible. Text, Info, Warning and Error. Following pragmas are used as example:

{text 'This is a Text message'}
{info 'This is an Info message'}
{warning 'This is a Warning message'}
{error 'This is an Error message'}

The pragmas can be located anywhere in a POU. In the Build log following output is visible:



In contrast to comments which are only somewhere in the code the message pragma is not ignored by the compiler and has an output in the Build Messages. This can be used for tagging ToDos, code to be checked or anything else which has to be change before releasing/ commissioning. This way a download of not working code or with intern comments can be prevented.

Once the POU is compiled it will not be compiled anymore as long as there are no changes in the POU or doing a rebuild all.

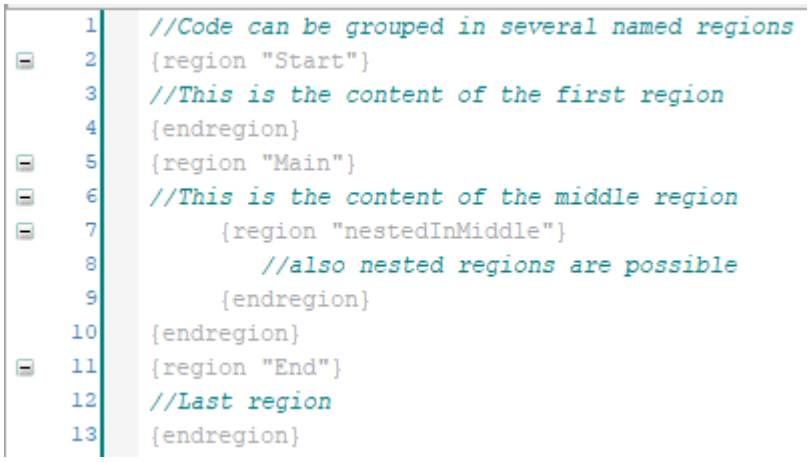
	Note: The message pragma cannot be used to output a message during runtime. To do a logging during runtime the CmpLog Library can be used in CoDeSys.

2.2 Region pragmas

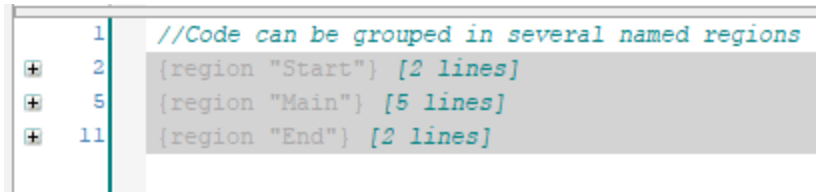
The region pragma can be used to group the code into several regions. This improves the code readability. Each region can be named. Also nested regions are possible. A region can be defined like shown below.

{region "<Name>"}
<Multiple lines Content>
{endregion}

The screenshot below shows the implementation in Automation Builder. Each region can be collapsed by clicking – or shown by clicking +



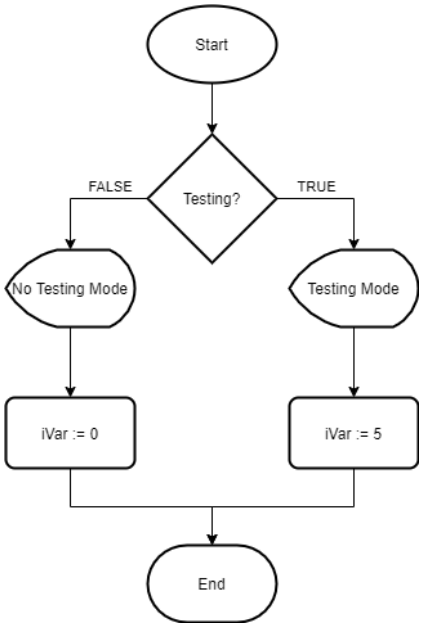
In the collapsed view it is visible how many lines are hidden in the region.



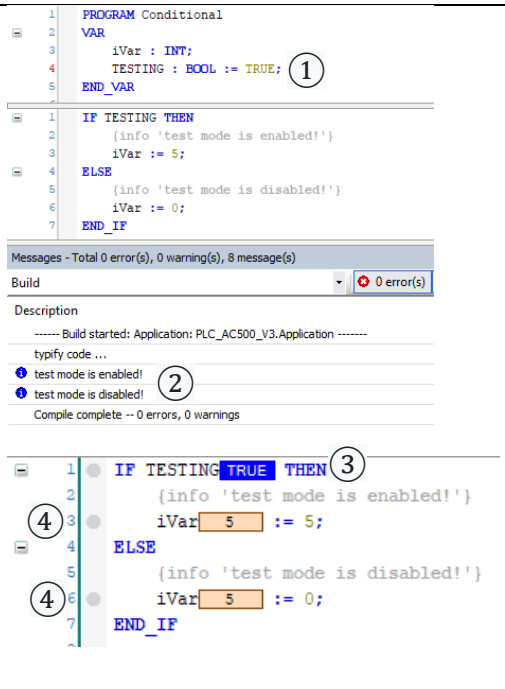
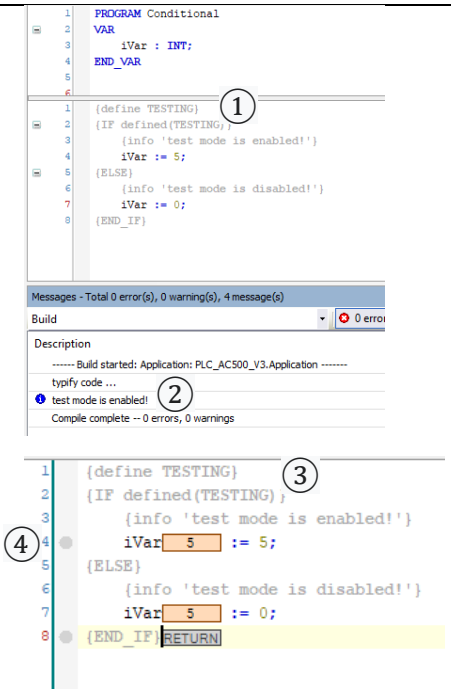
2.3 Conditional Pragmas

The conditional pragmas {IF} {ELSIF} {ELSE} {END_IF} can be used similar to the normal if statements. Usually if is used to check the state of a parameter which can be changed during runtime. The pragma {IF} checks identifiers which are not changing during runtime. This way the compiler result is different on the defined identifiers.

Following example is used. Depending on 'Testing' the variable is either set to 0 or 5. In addition, a message pragma is used.



The table below compares the code programmed with the classic if clause as well as using conditional pragmas.

	IF condition	Conditional Pragma
		
①	TESTING is a variable which is true	TESTING is an identifier wich is defined
②	The complete program is compiled both infos are outputted.	Only the TESTING part is compiled. Only the test mode is enabled info is outputted.
③	TESTING can be changed during runtime.	The TESTING identifier cannot be changed during runtime
④	Possible breakpoints at both assignments possible	Possible breakpoint only at the assignment in the first case.

Conditional Pragmas cannot be changed during runtime but using conditional pragmas instead of if conditions reduces the code size as well as the execution time. The conditional pragmas can be used as shown above to change the behavior pending on an identifier. This can for example be used to simulate signals or inputs that are not connected or change the behavior of the system pending on connected modules. In the real application the additional code has not to be deleted. Only the define has to be deleted or undefined.

Another use case is for example having two similar Applications. Both Applications should have the same source code. With ‘defined’ it is also possible to check if a variable or a type is defined. Furthermore, it can be checked if POUs, tasks or resources are existing in the application. Even more checks are possible. The complete list can be found in the online help. Pending on the available POUs different code will be compiled and downloaded.

2.4 Attribute Pragmas

Attribute pragmas is the biggest category. Only a few examples were chosen for this document. The complete list can be found in the online help.

2.4.1 DisplayMode

The attribute Display mode can be used to set the display mode of the following variable to decimal, hexadecimal or binary.

Like shown in the picture below 4 values are decelerated at the same memory address. So all values are the same.

```
VAR
    {attribute 'displaymode':='dec'}
    intDec          AT %MW0: INT;
    {attribute 'displaymode':='hex'}
    intHex          AT %MW0: INT;
    {attribute 'displaymode':='bin'}
    intBin          AT %MW0: INT;
    intNotDefined   AT %MW0: INT := 1000;
END_VAR
```

In the online mode the three variables are displayed in the defined display mode.

Expression	Type	Value	Prepared value	Address
intDec	INT	1000		%MW0
intHex	INT	16#03E8		%MW0
intBin	INT	2#0000001111101000		%MW0
intNotDefined	INT	1000		%MW0

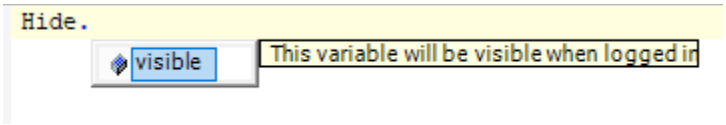
When changing the general display mode from decimal to any other only intNotDefined will change. The other values will remain.

2.4.2 Hide, hide all locals

The attribute ‘hide’ hides selected variables. This variable will not be visible in online mode or when accessing the POU variables from another POU.

```
VAR
    {attribute 'hide'}
    invisible :     BOOL;   // This variable will not be visible when logged in
    visible :       BOOL;   // This variable will be visible when logged in
END_VAR
```

In the Program hide two variables are define ‘invisible’ and ‘visible’. When accessing the POU ‘Hide’ only ‘visible’ is shown as member.



It is also not possible see the variables with the attribute hide in online mode or to monitor them.

In addition to the attribute ‘hide’ the attribute ‘hide_all_locals’ is available. This hides all local variables. Only input and output variables are visible then.

The attribute ‘hide’ is set in the line above the variable which shall be hidden.

The attribute ‘hide_all_locals’ is set in the very first line before the POU definition.

2.4.3 Initialize on call

The attribute pragma ‘initialize_on_call’ can be added to function block inputs. Each input which is not especially set during the call is initialized with the default value. This pragma can be used when inputs are not necessary for the function block. For example, an abort input.

```
inst(Input := iValue);
IF xAbort THEN
    inst(Abort := TRUE);
END_IF
```

Usually the function block is only called with an input variable. But somewhere in the code the abort value is set. Even when ‘xAbort’ is not set anymore and this code is not executed anymore the abort input of the instance will remain true.

This is especially critical when working with pointers to make sure that there is no invalid pointer at the function block input.

In the example below the Abort input has the attribute ‘Initialize on call’. In addition the whold function block must have the same attribute as definition above the POU definition.

```
{attribute 'initialize_on_call'}
FUNCTION_BLOCK initOnCall
VAR_INPUT
    Input : INT;
    {attribute 'initialize_on_call'}
    Abort : BOOL;
END_VAR
```

If now the code above is executed Abort will only be set to TRUE for the cycles where inst(Abort := TRUE); is called. Otherwise Abort will be initialized with FALSE again.

	Attention: When using ‘initialize_on_call’ an input assign before the function block call has no effect. So following input assign is not working when the attribute is used at the inputs. *Inst.A := 1;* *Inst.B := 2;* *Inst();*

2.4.4 Noinit

A program has two variables which are initialized with 1.

```
iVar1 : INT := 1;
{attribute 'noinit'}
iVar2 : INT := 1;
```

After login the two values are changed to 5.

When doing a reset warm or reset cold iVar1 is 1 again and iVar2 remains 5.

iVar1	INT	1
iVar2	INT	5

	Attention: A ‘noinit’ variable is not remanent. A download or reboot will set the variable back to 1 again.

2.4.5 No instance in retain

When developing a function block, it might be necessary to initialize some variables each time the function block is called. Therefore, the variables must not be retained. To prevent that a function block is instantiated in the retain area the attribute ‘no_istance_in_retain’ can be added in the line above the function block declaration. In case the user tries to instantiate this fb in the retain area the compiler outputs an error message.

2.4.6 Obsolete

A structure, function block, or other POU can be set to obsolete. If this object is compiled because it is instantiated or called in any POU or Task a warning is displayed.

It is recommended to use the attribute obsolete in case of maintaining old function blocks in the project or library. In case the wrong function block is instantiated the warning is outputted to the user.

2.4.7 Qualified only

The attribute ‘qualified_only’ can be found after adding a new Global Variables List. The position is in the line above VAR_GLOBAL. The effect is that global variables can only be accessed by using the list name as address. For example, ‘GVL.A’.

The following picture shows the Global Variables list ‘GVL’ which has the attribute qualified only. In this list is the variable ‘A’. Another Global Variables list ‘GVL_1’ doesn’t have the attribute qualified only. In this list are the variables ‘B’ and ‘C’.

The program itself has the local variable ‘C’.

When trying to access A the namespace GVL must be used. So the assign is ‘GVL.A := 3’.

‘B’ and ‘C’ can be accessed without the namespace as GVL_1 is not qualified only.

	Attention: Local variables shadow global variables. As ‘C’ is both a Global variable and a Local variable C := 3 will change the local variable. To change the global variable the address GVL_1.C has to be used even if the list is not ‘qualified_only’.

GVL

```
1  {attribute 'qualified_only'}
2  VAR_GLOBAL
3      A : INT := 2;
4  END_VAR
```

GVL_1

```
1  VAR_GLOBAL
2      B : INT := 2;
3      C : INT := 2;
4  END_VAR
```

qualifiedOnly

```
1  PROGRAM qualifiedOnly
2  VAR
3      C: INT := 2;
4  END_VAR
5
```

```
1  GVL.A := 3;
2  B := 3;
3  C := 3;
```

GVL

PLC_AC500_V3.Application.GVL

Expression	Type	Value
A	INT	3

PLC_AC500_V3.Application.GVL_1

Expression	Type	Value
B	INT	3
C	INT	2

qualifiedOnly

PLC_AC500_V3.Application.qualifiedOnly

Expression	Type	Value
C	INT	3

```
1  GVL.A  3  := 3;
2  B  3  := 3;
3  C  3  := 3;RETURN
```

2.4.8 Warning disable, warning restore

The compiler might throw warnings or errors during compilation. Warnings can be suppressed with a pragma. Following program is assigning an integer value to an unsigned integer value.

```
1  PROGRAM WarningDisable
2  VAR
3      iValue :    INT;
4      uiValue :   UINT;
5      uiValue2 :  UINT;
6  END_VAR
```

```
1  IF iValue <0 THEN
2      uiValue := 0;
3  ELSE
4      uiValue := iValue;
5      {warning disable C0195}
6      uiValue2 := iValue;
7      {warning restore C0195}
8  END_IF
```

Messages - Total 0 error(s), 2 warning(s), 0 message(s)

Build — 0 error(s) | 1 warning(s) | 0 message(s)

Description	Project	Object	Position
------ Build started: Application: PLC_AC500_V3.Application -------			
typify code ...			
C0195: Implicit conversion from signed Type 'INT' to unsigned Type 'UINT' : possible change of sign	Pragmas	WarningDisable [PLC_AC500_V3: PLC Logic: Application]	Line 4, Column 1 (Impl)
Compile complete -- 0 errors, 1 warnings			

In line 4 a warning for an implicit type conversation is thrown.

The same conversation is also done in line 6 but here is no warning outputted. The warning is suppressed by {warning disable <compiler id>}. This pragma can be used to suppress the warnings. After the assignment the warning is restored again. Please do not disable warnings before checking the impact of a faulty case. In this example the integer value is only assigned if it is greater than zero.

Attention: when disabling a warning please always restore the warning afterwards again. Otherwise the warning will also be disable in other program parts which are compiled later.

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