Motion Control Products

Application note
PLC coding style

AN00244

Rev A (EN)

A consistent approach to writing AC500 PLC programs, particularly applications involving motion control, improves quality, portability, readability and maintainability

Introduction

This application note describes ABB's recommended practices for coding ABB AC500 PLC applications. Adopting these practices ensures that application developers can write code with a consistent style thereby improving quality, portability, readability and maintainability.

This document assumes a working knowledge of ABB's Automation Builder software and is intended as a guide on how to structure the IEC 61131-3 application code and how to name application elements such as Program Organisation Units (POUs), program variables, constant values etc...

POU structure

The POU tab within the PLC application development environment allows the user to view the hierarchy of the application. Regardless of whatever method is used to structure the POUs within the POU tree view, a program named "prgProjectInfo" (containing comments only) should always be included at the outermost level of the POU tree. It is suggested that this is the only POU that is not contained within a folder.

This program should be excluded from the build (right click a POU and select 'Exclude from Build') and this will be indicated by its green colour in the POU tree.

The comments in this program are used to detail the application's function and revision history. It is recommended that this program be coded using Structured Text (ST). The program contents should contain the following information as a minimum…

- Project reference
- Customer
- Hardware platform(s) used including firmware revision details (hardware list should include additional devices other than the PLC that are in some way controlled by the PLC, e.g. servo drives, fieldbus couplers)
- Author
- Date of creation
- Short description
- Revision history (revision number, date of change, author, description of change)

The screenshot below illustrates the general form for this program (an Export file is included with this application note to allow the user to import this into their project if required)…

prgProjectInfo (PRG-ST)

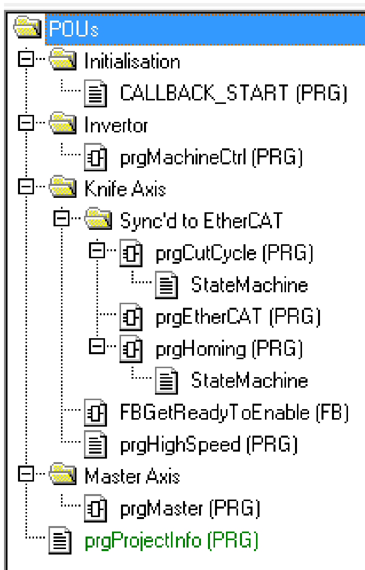
```
0001 PROGRAM prgProjectInfo
0002 VAR
0003 END_VAR
0004
```

```
0001 (* ======================================================================
0002     Project        :   ProjectReferenceNumber
0003     Customer       :   ABB Valued Customer
0004     H/W Platform   :   PM5xx (2.5.2)
0005                        CM579-ETHCAT (4.3.0.2)
0006                        MicroFlex e150 (5852)
0007                        MotiFlex e180 (5860)
0008     Author         :   ABB UK Ltd
0009     Date           :   7th June 2016
0010     Description    :   Short form project description
0011 ======================================================================
0012     Revision History
0013
0014     Version    Date        Author    Description
0015     x.x.x      xx/xx/xxxx  xx        Short description of changes
0016
0017
0018 ======================================================================*)
0019
```

For the remaining POUs the general recommendation is to use folders (and sub-folders) in a way that simplifies the apparent structure of the code and allows someone who has not seen the application before to understand the logical structure of the code and easily locate relevant POUs when editing/debugging the application. How this is actually achieved may depend on a number of factors including size/complexity of the application so this document does not aim to define a specific style. Instead two suggested schemes are illustrated below for reference.

Physical view

This scheme assumes that the user would prefer to structure the POU view in a way that relates to the physical structure of the machine/system the code is targeted at. Top level folders should be used to contain POUs relating to specific areas/parts of the machine/system and if necessary sub-folders within these top level folders can be used to break the application down into further logical sections.

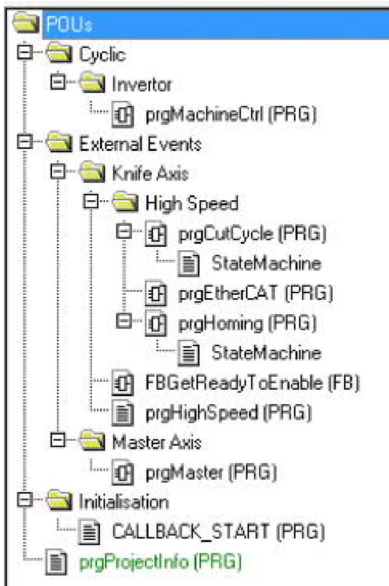


Note that there are some cases where POUs cannot be wholly “assigned” to a specific area of the machine via a folder structure. As an example, CALLBACK_START is a program that is called by the system start event and may contain logic relating to many separate elements of the machine. This program could be written to call additional programs that are then located within the relevant folders in the tree, but it is also acceptable to retain all of the logic within a single program to minimise the number of POUs and therefore the apparent complexity of the application. In this case a folder independent of the machine structure may be used (as illustrated by the Initialisation folder in the example to the left).

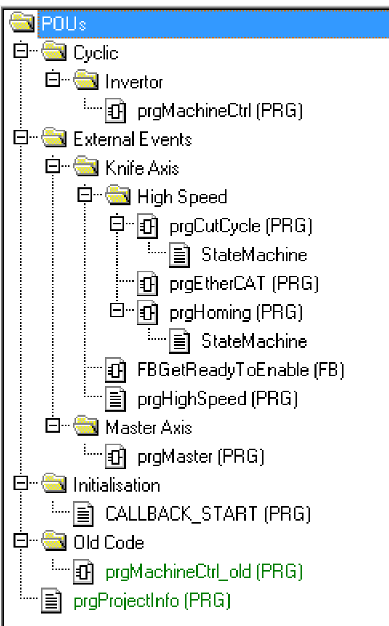
Also note that CALLBACK_START is a program but is not pre-fixed by the text ‘prg’. This particular program (and others that can be called by System Events) must use this specific name so cannot meet the recommendations of this document for program name pre-fixing.

Logical view

This scheme assumes that the user would prefer to structure the POU view in a way that relates to the logical processing of the application. Top level folders should be used to contain POUs relating to specific logical functions and these should be named to assist in identifying how/when the POUs within the folder are processed (e.g. if a task has been configured to run a number of programs on a cyclic basis a top level folder named 'Cyclic' may be created). If necessary sub-folders within these top level folders can be used to break the application down into further logical sections.



As can be seen from the two examples above, in practise a mix of the two schemes may be inevitable. Folders are sorted in alpha-numeric order so you may also make use of this fact should you wish to order your folders in a specific manner (e.g. starting Folder names 01…. 02…. etc… can allow a specific order to be defined in the POU tree). Folder names may also contain a space (unlike actual POU elements) so include spaces when defining folder names if required.



It may also be useful to include a folder (e.g. named 'Old Code', 'Unused' or other name with similar meaning) in which unused, outdated or trial elements of code that are now excluded from the build are stored for future reference. During application development it may be preferable to keep old versions of particular POUs to revert back to until you are sure the new application code is working correctly.

General naming rules for POU elements

The following general rules/conventions apply to all POU elements:

- POU's (PRGs, Function Blocks, Functions) are automatically sorted in alpha numeric order
- POU name lengths are limited to a maximum of 255 characters (it is recommended to try to restrict lengths to 25 characters or less if possible)
- POU names cannot start with a number
- No spaces are allowed within POU names
- Underscores '_' are allowed but it is recommended not to use these but instead use lower case prefixes followed by UpperCamelCase, also known as "PascalForm" (i.e. capitalised first letter of each word within the name)
- The order in the POU tree view has no bearing on the order the POU's are called, it is just a list

Programs

When the user declares a POU of type 'Program' it is recommended that the prg prefix is included in the name. This allows the user to easily identify programmatic access to a member variable of a program and distinguish this from a member variable of a data structure for example. The remainder of the name should use UpperCamelCase.

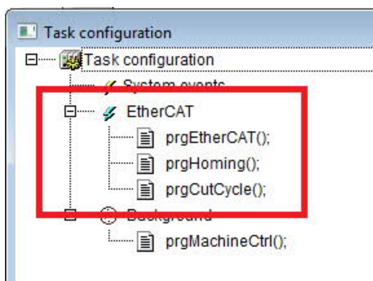
Prefix	Type/Description	Example
prg	Program	prgMachineControl

Avoid meaningless names such as prg01 and instead select a name that relates to the program's purpose (e.g. prgMachineControl may identify that this POU was the main program containing the bulk of the machine logic.

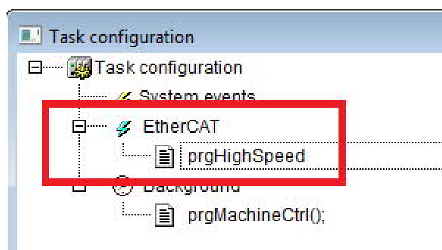
It is advised that as few programs as possible are called directly from tasks. Instead it is best that a program called from a task is then used to call all other programs that need to be processed at the priority defined by the calling task/program.

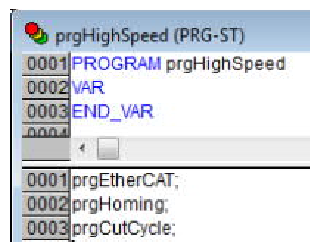
Example:

Instead of….



…use….





Actions

Action names should not be prefixed. Use UpperCamelCase for the action name. Avoid meaningless names (e.g. Action01).

Prefix	Type/Description	Example
(none)	Action	StateMachine

Variables and Constants

All program elements, including variables and constants should have meaningful names. Whenever possible, do not use hard-coded physical addresses as variables, always define a variable via the IO mappings screen(s) in Automation Builder. Do not re-use a global variable name within a local routine (so called "shadowing"). All variables should have suitable prefixes to indicate their type/usage as described in the table below. Constants follow a similar scheme but the prefix is preceded by an underscore to indicate a constant/literal value.

The main body of the name should describe the use of the variable/constant and should be mixed case, capitalizing the first letter of each word with no spaces or underscores. This format is known as UpperCamelCase or PascalForm. Examples are shown in the following table…

Prefix	Type/Description	Example
x	BOOL – Boolean, 1 bit data (true/false)	xMyBool: BOOL;
by	BYTE – Byte, 8 bit data	byMyByte: BYTE;
w	WORD – word, 16 bit data	wMyWord: WORD;
dw	DWORD – double word, 32 bit data	dwMyDoubleWord: DWORD;
lw	LWORD – long word, 64 bit data	lwMyLongDoubleWord: LWORD;
si	SINT – small integer, 8 bit data	siMySmallInteger: SINT;
usi	USINT – unsigned small integer, 8 bit data	usiMyUnsignedSmallInteger: USINT;
i	INT – integer, 16 bit data	iMyInteger: INT;
ui	UINT – unsigned integer, 16 bit data	uiMyUnsignedInteger: UINT;
di	DINT – double integer, 32 bit data	diMyDoubleInteger: DINT;
udi	UDINT – unsigned double integer, 32 bit data	udiMyUnsignedDoubleInteger: UDINT;
li	LINT – long integer, 64 bit data	liMyLongInteger: LINT;
uli	ULINT – unsigned long integer, 64 bit data	uliMyUnsignedLongInteger: ULINT;
r	REAL – real, 32 bit data	rMyReal: REAL;
lr	LREAL – long real, 64 bit data	lrMyLongReal: LREAL;
str	STRING – string, string type data	strMyString: STRING;
t	TIME – time	tCurrentTime: TIME;
tod	TIME_OF_DAY – time of day	todTimeOfDay: TIME_OF_DAY;
dtt	DATETIME – date time	dttCurrentDateAndTime: DATETIME;
dt	DATE – date	dtCurrentData: DATE;
p	POINTER	pMyPointer: POINTER TO DINT;
ar	ARRAY	arMyArray: ARRAY[1..5] OF INT;
E	Enumeration Type definition	TYPE ECutterState
e	Enumeration instance and values	eState : ECutterState; TYPE ECutterState (eInitState := 0, eHome := 10, eHomed := 20); END_TYPE
TS	TYPE STRUCT definition	TYPE TSMyTypeStructure
ts	Type Struct instance	tsLabelHead1 : TSLabelHead;
ST	STRUCT definition	STRUCT STMyStructure
st	STRUCT instance	stStructure: STMyStructure;
_	Constant Value	_iMaxSpeed: INT:= 3000;
ax	Axis Reference	axCutter: AXIS_REF;

This naming convention for variables also applies to PDO mapped variables. For example, an EtherCAT axis 'ControlWord' PDO mapping is a U16 data type (word) so a suitable name for this could be wAxisControlWord.

Functions

Functions come in two main forms. They can be standard functions which are included within the IEC61131 programming environment libraries or they can be user defined. When a user defined function POU is created it should be given a name that describes the basic functionality of the function. The name should be prefixed with 'do' and the remainder of the name should be of UpperCamelCase form.

Prefix	Type/Description	Example
do	FUNCTION	doWrapValueToRange

Once a Function has been created or is used from a library it does not have instances and as such no rules are needed for handling its repeated use throughout the code.

Function blocks

Function blocks come in two main forms. They can be standard function blocks which are included within the IEC61131 programming environment libraries or they can be user defined function blocks.

Function block declarations

When a function block has been used from a standard library then it already has its name and functionality defined (and this will not include a prefix). When a user defined function block POU is created it should be given a name (of UpperCamelCase form) that describes the basic functionality of the function block and this should be prefixed with the capital letters FB.

Prefix	Type/Description	Example
FB	FUNCTION BLOCK	FBPhaseAxis

Instances of function blocks

Once a function block has been created, or is used from a library, then it will be necessary to define a name for any instance of the function block that is to be used. When defining an instance name start this name with a short form of the function block type in lower case (2 or 3 characters ideally) followed by a description of its use in UpperCamelCase as shown in the examples below…

Prefix	Type/Description	Example
pha	FBPhaseAxis	phaInfeedAxis
par	CMC_AXIS_CONTROL_PARAMETER_REAL	parMainDrive
ker	CMC_MOTION_KERNEL_REAL	kerMainDrive
cia	ECAT_CiA402_CONTROL_APP	ciaMainDrive
mr	MC_MoveRelative	mrFirstIndex
ton	T_ON: Timer	tonStartDelay
ctu	CTU: Counter	ctuProductionCount

Avoid names such as ctu1 for example. Instead use a name relative to the block's purpose (e.g..ctuServiceDue if it were counting machine cycles to indicate when a machine service was required). Try to keep to two or three letters for the prefix, but this may not always be possible (e.g. it might be difficult to avoid a clash with another function block prefix or difficult to make it obvious what the function block type is), in which case try not to exceed five characters.

As an example, both of the following instance names would be acceptable…

```
phaInfeedAxis : FBPhaseAxis;
phaseInfeedAxis : FBPhaseAxis;
```

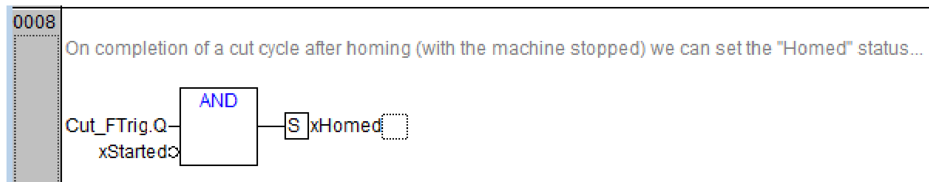
…but the five letter version is maybe a little clearer to the reader.

Comments / Documentation

There are three main methods of entering comments depending on the specific IEC61131-3 programming language used and/or the area of the IEC 61131 program editor the comments are entered into.

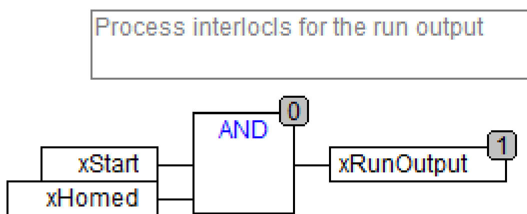
In Structured Text (ST), Instruction List (IL) and the variable list areas, comments are encapsulated with (* Comment here *).

Example:



However when Ladder (LD) or Function Block Diagram (FBD) are used comments can be added to each individual network by right-clicking and selecting 'Comment' from the menu presented.

Example:

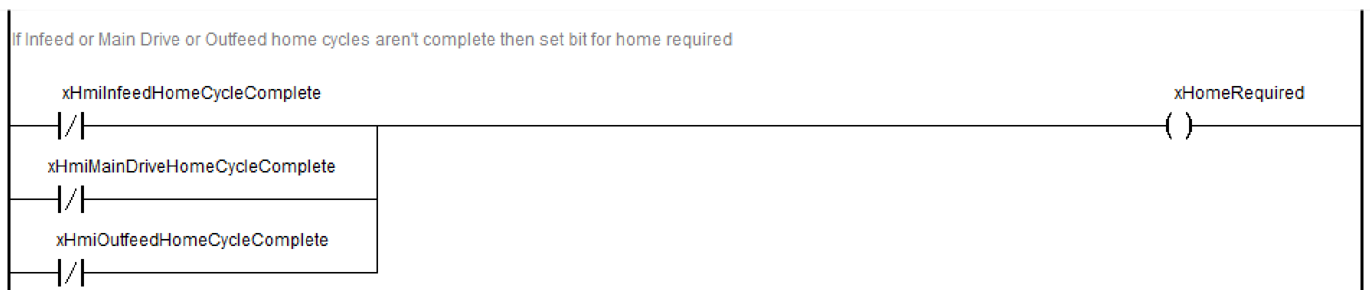


Continuous Function Chart (CFC) uses a comment box feature which can be dropped on to the page.

Process interlocls for the run output
AND
0
xStart
xHomed
xRunOutput
1

Whichever comment method is used program comments should describe what is happening and how it is being done. Avoid comments where functionality is clear from the code.

For example, avoid…

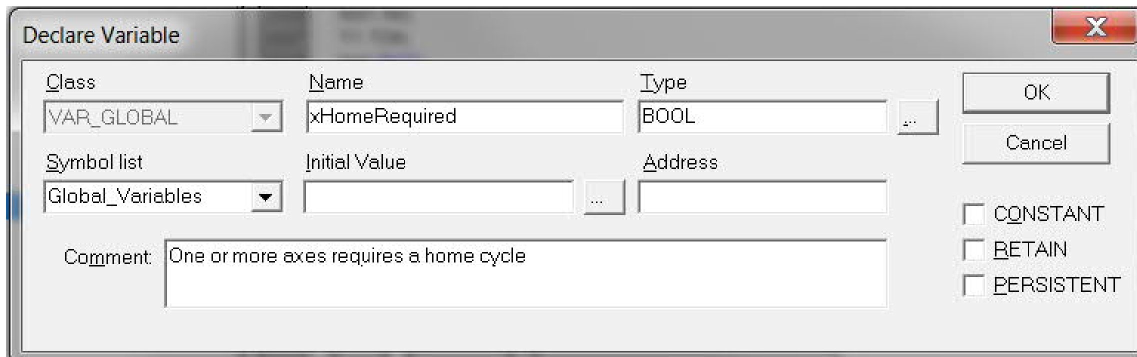


…and instead use a comment of the form…

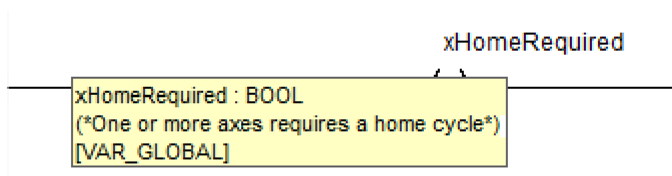
Or …

```
(*Check if home cycles are complete*)
xHomeRequired:= NOT xHmiInfeedHomeCycleComplete AND NOT xHmiMainDriveHomeCycleComplete AND NOT xHmiOutfeedHomeCycleComplete;
```

Comments can also be added to individual variables via the declaration window that pops up when the variable is first created…



This means that as a user hovers over the variable they will not just see the properties of the variable, but also its comments;



Contact us

For more information please contact your local ABB representative or one of the following:

new.abb.com/motion
new.abb.com/drives
new.abb.com/drivespartners
new.abb.com/PLC

© Copyright 2016 ABB. All rights reserved.
Specifications subject to change without notice.