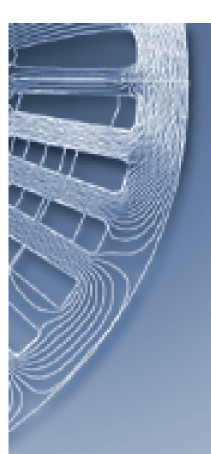
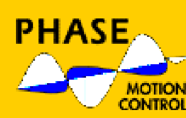





# AX-V

# PROGRAMMING MANUAL

# Global Programmable Logic Controller

Release 2.1

Date 10-12-2001

**Supported Models:**

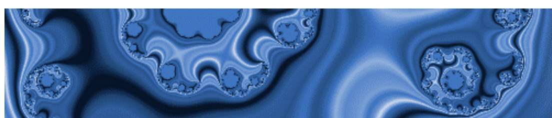
AX-V family drives

**Configuration tool:**

AXV Cockpit

---

Phase Motion Control s.r.l.
Lungobisagno Istria, 14D/27
16141 Genova – Italy
Tel. +39 (010) 8359001
Fax +39 (010) 8355355
e-mail: support@phase.it



# 1. INDEX


1. INDEX ........ 2

2. INTRODUCTION ........ 4
   GENERAL FEATURES ........ 4

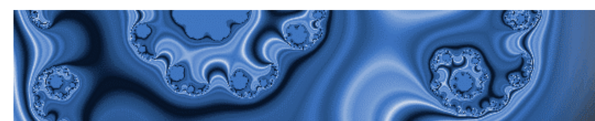
3. LANGUAGE ........ 4
   SOURCE ........ 4
   BASIC ELEMENTS ........ 5
   DATA, VARIABLE AND CONSTANT TYPES ........ 5
   PROGRAMS ........ 8
   BASIC INSTRUCTIONS ........ 8
   FUNCTION ........ 11
   FUNCTION BLOCKS ........ 12
   STANDARD FUNCTIONS ........ 15
      SHL, SHR, ROL, ROR functions ........ 16
      SEL function ........ 16
      MUX function ........ 16
      MAX, MIN functions ........ 16
      LIMIT function ........ 17

4. GPLC PROJECT ........ 17
   DESCRIPTION ........ 17
   GPLC PROJECT COMPONENTS ........ 17
      Source modules ........ 17
      IMG Files ........ 18
      TASK ........ 18
   CREATING AND RUNNING A GPLC PROJECT ........ 18

5. PROGRAM DESCRIPTION ........ 19
   FEATURES ........ 19
   EDITOR ........ 20
   COMPILER ........ 20
   COMMUNICATION INTERFACE ........ 20
   DATA MONITOR ........ 21

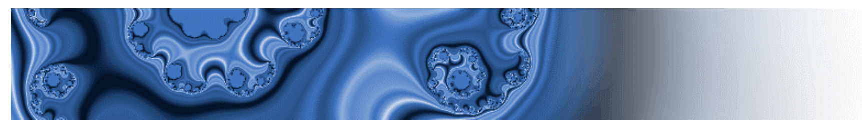
6. PARAMETERS INTERFACE ........ 22
   INTRODUCTION ........ 22
   PARAMETERS and VARIABLES DECLARATION ........ 22

7. FIRMWARE DRIVE INTERFACE ........ 25
   INPUT, OUTPUT AND DATA BLOCKS ........ 25
   FIRMWARE INTERFACE FILES ........ 26
   DIGITAL I/O ........ 26
   ANALOG I/O ........ 27
   ENCODER SIGNALS READING ........ 27
   INCREMENTAL ENCODER AUTOMATING PHASING ROUTINE ........ 30
   ENCODER SIGNALS REPETITION ........ 30



CURRENT LOOP ... 31
POSITION/VELOCITY LOOP ... 32
DSP CONTROL BIT ... 34
BLOCK DIAGRAM ... 35
Symbols ... 35
Overview ... 36
Current loop ... 36
Velocity/position loop ... 37
Ramp generator ... 37

8. APPLICATION EXAMPLE ... 38
DESCRIPTION ... 38
CREATING A PROJECT ... 38
INIT PROGRAM (file basicinit.plc) ... 40
SLOW PROGRAM (basicslow.plc) ... 41
FAST PROGRAM (basicfast.plc) ... 42
VARIABLE AND PARAMETERS (basicpar.plc) ... 43
AX-V COCKPIT PARAMETERS TABLE ... 43
COMPILE THE PROJECT ... 43



# 2. INTRODUCTION

*GENERAL FEATURES*

GPlc is an application program designed for Windows 95/NT Operating Systems which can create PLC programs for AX-V family drivers.

The main program elements are:

- Integrated text editor for PLC program editing.
- PLC language source module compiler.
- Communication interface to download the PLC code generated by AXV driver compiler.
- Watch window to view the variables used by PLC program.

More details are given in the following paragraphs.

# 3. LANGUAGE

*SOURCE*

All the instructions and structures foreseen in the GPlc language are in accordance with the IEC1131-3 standard, the relative IL (instructions list) is integrated in the language.

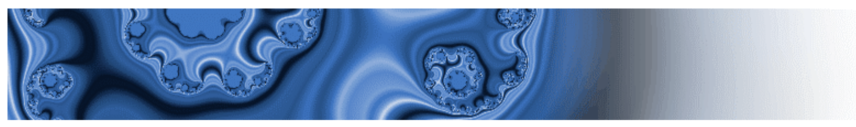
So far, the included elements are:

- All data types use except for LINT, REAL, STRING and TIME.
- Variables and their data type definition.
- Variables single dimension arrays definition.
- Variables and relative attributes GLOBAL, CONST, RETAIN, AT declaration and initialization structure.
- Programs and variables declaration within the programs themselves.
- IL instructions set.
- Standard functions set (except for string).

A brief description of the language elements implemented is reported here after. For a detailed definition, please refer directly to the standard IEC 1131-3.

Conventions used in the present description:

- The language elements are printed in a regular courier font .
- The elements showing names and types assigned by the programmer are printed in *italic courier font.*
- The optional elements of any structure are reported between italic square brackets.



## *BASIC ELEMENTS*

- The source modules are edited using standard ASCII characters.
- The addition of comments between (* and *) is possible in any point of the source modules.

## *DATA, VARIABLE AND CONSTANT TYPES*

### Data types

Defined data types :

| Keyword | Data type | Bits | Range |
|---|---|---|---|
| BOOL | Boolean | 1 | 0 ÷ 1 |
| SINT | Short integer | 8 | -128 ÷ 127 |
| USINT | Unsigned short integer | 8 | 0 ÷ 255 |
| INT | Integer | 16 | -32768 ÷ 32767 |
| UINT | Unsigned integer | 16 | 0 ÷ 65536 |
| DINT | Double integer | 32 | $-2^{31} \div 2^{31}-1$ |
| UDINT | Unsigned long integer | 32 | $0 \div 2^{32}$ |
| BYTE | Bit string of 8 | 8 | X |
| WORD | Bit string of 16 | 16 | X |
| DWORD | Bit string of 32 | 32 | X |
| REAL* | Single precision floating point | 32 | |

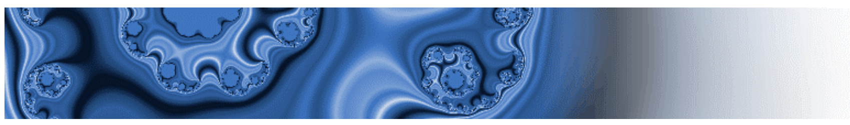
* this type of variables cannot be used in the Fast Task

### Declaring variables

The variables declaration is done using one of these structures:

```
VAR_GLOBAL [RETAIN][CONST]                    VAR [RETAIN][CONST]
        .                                             .
        .                                             .
variables list                  or            variables list
        .                                             .
        .                                             .
END_VAR                                       END_VAR
```

- Any variable used by all the defined programs inside the application is declared using the VAR_GLOBAL .. END_VAR structure.
- Any variable used by a single defined program inside the application is declared using the VAR.. END_VAR structure.



- The CONST attribute defines variables within a structure with constant and unchangeable value.
- The RETAIN attribute defines variables keeping their value, also after the driver reset or switch off.

Variables inside the structures can be declared with following statements:

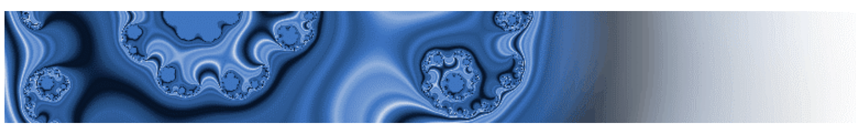
*VariableName* : *DataType [* := *Startvalue ]*;

*VariableName* : ARRAY OF[ 0..*n* ] OF *DataType*;

*VariableName* AT *Location* : *DataType [* := *StartValue ]*;

where:

- *VariableName* is an alpha-numeric string identifying the variable.
- *Datatype* is one of the types foreseen (see the relative table).
- *StartValue* is the variable value after the system reset.
- *Location* is a logical address defined inside the driver firmware (see the following description ).



Example:

```
PROGRAM test

        VAR
                QuoteX : DINT;
                Enable : BOOL := FALSE;
                Counters : ARRAY[ 0..10 ] OF UINT;
                CurrentTask AT %MW4.32 : INT;
        END_VAR
        .
        .
        instructions
        .
        .
END_PROGRAM
```

This structure declares four local variables ( within the program ). After reset, the variable Enable is set to FALSE, the variable Counters is an array of 11 variables type unsigned int, the variable CurrentTask is the integer defined in the drive memory at block 4, index 32.

## Array

As mentioned before, the use of variables array is possible. Any element of the array can be accessed using the array variable name followed by the index between square brackets.
A variable name can also be used as index:

```
LD      Quote[ 7 ]
ST      Posit[ idx ]
```

## Location

The key word AT, in the variables declaration, is used to define a variable as the value in a specified address of the memory driver.
This variable type allows the access to any variable defined inside the driver firmware (see paragraph 0 and 6).
The location is defined as follows:

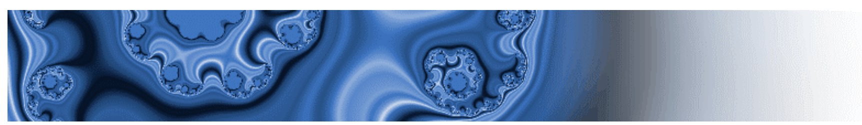
*%location dimens index.index ..*

where *location* and *dimens* can have following values:

| Location | |
|---|---|
| I | Input location |
| Q | Output location |
| M | Memory location |

| Dimension | |
|---|---|
| X | 1 bit dimension |
| B | 8 bit dimension |
| W | 16 bit dimension |
| D | 32 bit dimension |

The points (" .") after the index specify the position in the indicated area.



Example:

```
%MW4.6        (* Memory word block 4 index 6 *)
%IX0.4 (* Input bit set 0 index 4 *)
```

## Constant

Following types are foreseen:

| Type | Statements | Example |
|---|---|---|
| Boolean | TRUE, FALSE | |
| Decimal | decimal digit | 534 -8000 ecc. |
| Hexadecimal | prefix 16# followed by hexadecimal digit | 16#7A22 |
| Octal | prefix 8# followed by octal digit | 8#302 |
| Binary | Prefix 2# followed by 0, 1 digit | 2#11001010 |

## *PROGRAMS*

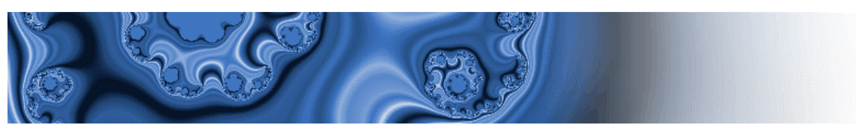
A PLC code executive unit is declared as a program using the structure PROGRAM..END_PROGRAM.
According to the IEC standard, a PLC program: is identified by a name, can contain more variables declarations structures and groups together a list of instructions that can access the local and global variables.
Each program is linked to an executive task of the host machine.
The declaration structure is the following:

```
PROGRAM Programname
        .
        variables declaration
        .
        .
        instructions list
        .
END_PROGRAM
```

## *BASIC INSTRUCTIONS*

The language used is IL (Instruction List).

- An instruction list is composed of an instructions sequence.



- Each instruction starts in a new text row and is composed by an operator that can be optionally followed by operator modifiers and by one or more operands separated by commas.
- Operands can be variables, constants or labels.
- A label can optionally precede each instruction.
- Comments can be included everywhere in the instruction list using (* for the beginning and *) for the ending of a comment.

Example:

```
Beginning:                      (* Sequence beginning point *)

        LD      inp0            (* Active if input 0 *)
        ANDN    alarm           (* and no alarm *)
        ST      start           (* begin cycle *)
```

The IL language uses an accumulator register (or *current result* as defined by the IEC standard) that stores the last executed operation result.
The accumulator is the first operand of each instruction, the other possible operands follow the instruction.

Instruction example:

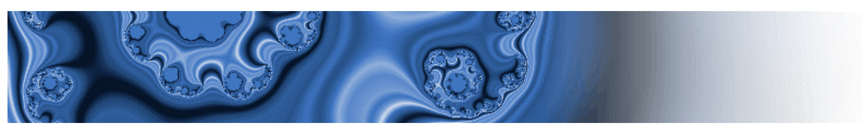
```
        AND     varX
```

Means:

*accumulator* = *accumulator* AND varX

The complete standard instructions set is the following:

| Operator | Modifier | Operand | Function |
|---|---|---|---|
| LD | N | all | Stores operand into accumulator |
| ST | N | all | Stores accumulator into operand |
| S | | BOOL | Set TRUE the operand if accumulator is TRUE |
| R | | BOOL | Set FALSE the operand if accumulator is TRUE |
| AND | N, ( | all | Boolean AND or bit by bit between accumulator and operand |
| & | N, ( | all | Boolean AND or bit by bit between accumulator and operand |
| OR | N, ( | all | Boolean OR or bit by bit between accumulator and operand |
| XOR | N, ( | all | Boolean XOR or bit by bit between accumulator and operand |
| ADD | ( | All except BOOL | Addition |



| SUB | ( | All except BOOL | Subtraction |
|---|---|---|---|
| MUL | ( | All except BOOL | Multiplication |
| DIV | ( | All except BOOL | Division |
| GT | ( | All except BOOL | Comparison > |
| GE | ( | All except BOOL | Comparison >= |
| EQ | ( | All except BOOL | Comparison = |
| NE | ( | All except BOOL | Comparison <> |
| LE | ( | All except BOOL | Comparison <= |
| LT | ( | All except BOOL | Comparison < |
| JMP | C,N | label | Jump to label |
| ) | | | The delayed operation is executed |

The foreseen modifiers are C, N and (, they mean:

- C the instruction will be executed only if the accumulator is boolean TRUE.
- N the operand (BOOL) is reversed before to be used in the operation.
- "(" the operation execution has to be delayed until the operator ")".

Examples:

The instruction

JMPC beginning

means that the jump to the label beginning is done only if the accumulator is TRUE,

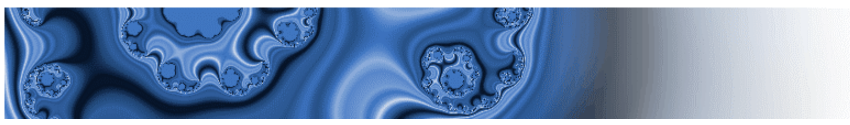
JMPCN beginning

means that the jump to the label beginning is done only if the accumulator is FALSE.
The instruction :

ANDN alarm

is interpreted as:

*accumulator* = *accumulator* AND NOT alarm



The sequence:

```
AND(   inp0
OR     inp1
)
```

is interpreted as:

*accumulator* = *accumulator* AND ( inp0 OR inp1 )

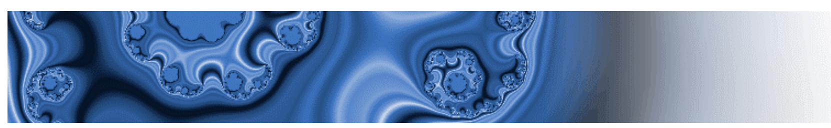
## *FUNCTION*

The construct FUNCTION..END_FUNCTION allow to declare a block of GPLC code defined as *Function*.
A *Function* is charcterised by a name, a list of input parameters and by the type of output data.

- It is possible to declare local variables inside a function. These variables are visible only inside the the function itself.
- Local variables do not conserve their value with two consecutive function calls
- The structure of a *Function* is as follws:

```
FUNCTION nomeFunzione : TypeOfReturnValue

        VAR_INPUT

                Input variables declaration

        END_VAR

        VAR

                Local variables declaration

        END_VAR

        Instruction list

END_FUNCTION
```

- The result of the function must be stored into a variable with the same name as the Function itself
- It is possible to use the instruction RET inside a function (with associated modifiers C and N) to execute a return on condition to the invoking program
- A *Function* cannot acces to global variables (included system variables)
- A *Function* is called from the main program placing the name of the function itself in the list of instructions
- When a *function* is called the first parameter passed is the value of the accumulator; additional parameters should follow the *function* name separated by colon.
- The output value of the *function* is placed in the accumulator
- A *function* can call another *function*



Example: Following function returns the square of a 16 bit (the return value is 32 bit)

```
FUNCTION Pow2 : DINT

      VAR_INPUT

            Val : DINT;

      END_VAR

      LD    Val
      LE    16#8000     (* Check if max value is exceeded *)
      JMPC  lExeMul

      LD    -1          (* Conventional value to indicate an error*)
      ST    Pow2
      RET

lExeMul:

      LD    Val         (* calculate square value *)
      MUL   Val
      ST    Pow2        (* store result into output varible *)

END_FUNCTION
```

Following example shows how to call the function Pow2 from the main code.

```
      .
      .
      LD    x
      Pow2        (*    Passaggio di X e invocazione funzione     *)
      EQ    -1    (*    Verifica risultato      *)
      JMPC  lErr
      .
      .
```

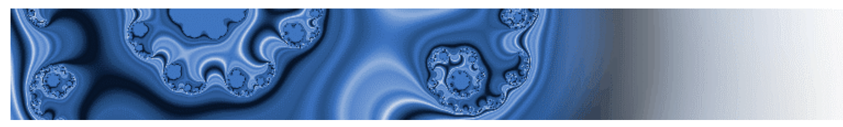
Following example shows how to call a function (Func) with more than one input variables from the main code.

```
      LD    x
      Funz  y, z  (*    Passing parameters di X, Y, e Z and call *)
                  (*    function Funz                            *)
      ST    ris   (*    Storing of return value into variable RIS*)
      .
      .
```

*FUNCTION BLOCKS*

The construct FUNCTION_BLOCK..END_FUNCTION_BLOCK allow to declare a code block defined as *function block*.

- A *function block* (according to IEC standard) is identified by a name and can use one or more input and output variables
- Several local variables can be declared inside a function block.



- A *function block* can access global variables only if they are declared with a dedicated construct VAR_EXTERNAL .. END_VAR inside the *function block* itself.
- The structure of a Function Block is as follows:

```
FUNCTION_BLOCK FunctionBlockName

      VAR_INPUT
            .
            Input variables declaration
            .
      END_VAR

      VAR_OUTPUT
            .
            Output variables declaration
            .
      END_VAR

      VAR_EXTERNAL
            .
            Declaration of global variables used by the function
            block
            .
      END_VAR

      VAR
            .
            local variables declaration
            .
      END_VAR
      .
      Instruction list
      .
END_FUNCTION_BLOCK
```

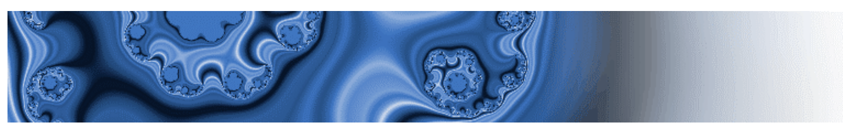
- To use function blocks, they must be declared inside the program where thhey are invoked.
- Several instances of the same function block can be declared in the same program (instances are distinguished by different names).
- Each instance must be defined in the variables declaration construct VAR..END_VAR in the same way as variables:

*InstanceName* : *FunctionBlockName*;

- Local variables declared inside a function block conserve their value at two consecutive calls of the same instance of the function block.
- Values of input and output variables are transferred to and from a function block with load and store operation as follows:

*InstanceName*.*VariableName*

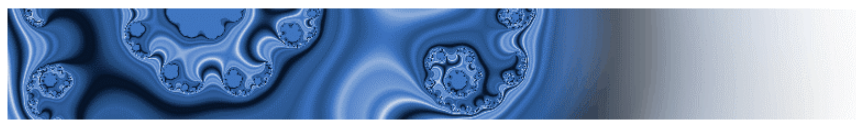
Example:



```
(*    This function block detect the rising edge of the input    *)
(*    variable INP                                    .          *)

FUNCTION_BLOCK RisingEdge

      VAR_INPUT
            Inp : BOOL;             (* Input variable *)
      END_VAR

      VAR_OUTPUT
            Edge : BOOL;            (* Output variable *)
      END_VAR

      VAR
            Memory : BOOL := TRUE;  (* memory of the input variable *)
      END_VAR

      LD    Inp
      ANDN  Memory
      ST    Edge

      LD    Inp
      ST    Memory

END_FUNCTION_BLOCK
```

## Example of program code using the RisingEdge functionblock

```
PROGRAM

      VAR
            Inp0 AT %IX0.0 : BOOL;  (*    Digital I/Os           *)
            Inp1 AT %IX0.1 : BOOL;  (*    of the drive           *)
            Out0 AT %QX0.0 : BOOL;
            Out1 AT %QX0.1 : BOOL;

            ReInp0 : RisingEdge;    (* Two instances of the      *)
            ReInp1 : RisingEdge;    (* same function block       *)
      END_VAR

      LD    inp0              (* Set of digital output 0 when      *)
      ST    ReInp0.Inp        (* a rising edge of inp0 is detected *)
      CAL   ReInp0
      LD    ReInp0.Edge
      S     out0

      LD    inp1              (* Set of digital output 1 when      *)
      ST    ReInp1.Inp        (* a rising edge of inp1 is detected *)
      CAL   ReInp1
      LD    ReInp1.Edge
      S     out1

END_PROGRAM
```



## *STANDARD FUNCTIONS*

Besides a basic instructions set, GPlc provides a set of standard functions foreseen by the IEC standards.
Standard functions list:

| Name | N° operand | Operand type | Returned type | Function |
|---|---|---|---|---|
| ABS | 0 | X | Accumulator type | Absolute value |
| MOD | 1 | all except BOOL | Accumulator type | Reminder after division |
| NOT | 0 | X | Accumulator type | Reverse |
| SHL | 1 | all except BOOL | Accumulator type | Left binary shift (*) |
| SHR | 1 | all except BOOL | Accumulator type | Right binary shift (*) |
| ROL | 1 | all except BOOL | Accumulator type | Left binary rotation (*) |
| ROR | 1 | all except BOOL | Accumulator type | Right binary rotation (*) |
| SEL | 2 | all except BOOL | Operand type | Selector (*) |
| MUX | n | all except BOOL | Operand type | Multiplexer (*) |
| MAX | n | all except BOOL | Operand type | Largest value (*) |
| MIN | n | all except BOOL | Operand type | Smallest value (*) |
| LIMIT | 2 | all except BOOL | Operand type | Limit between largest and smallest value (*) |

(*) see detailed description below .

The calling of a function is done specifying its name in the operand field, followed by possible arguments separated by commas. The accumulator value is used as the first function argument.
For example, the instruction:

ABS

is interpreted as:

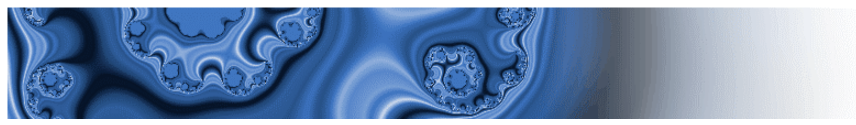
*accumulator* = ABS *accumulator*

The instruction:

MAX var1, quoteX, quoteY, 45

is interpreted as:

*accumulator* = MAX( *accumulator*, quoteX, quoteY, 45 )



## SHL, SHR, ROL, ROR functions

They shift or rotate the accumulator bits to the right or to the left as much as indicated in the operand field. The returned value is entered in the accumulator.
The instruction:

```
SHL     var
```

shifts the accumulator bits to the left of var positions.

## SEL function

SEL sets the accumulator to the same value as one of the two operand, depending on the accumulator boolean value. If the accumulator is FALSE, the first operand is entered in the accumulator; if the accumulator is TRUE, the second operand is entered in the accumulator.
The instruction:

```
LD      flag
SEL     QuoteX, QuoteY
```

enters the QuoteX value in the accumulator if the variable flag is FALSE before the instruction execution.

## MUX function

MUX is similar to a SEL instruction with the possibility to select between one or more operand values depending on the accumulator.
The accumulator numeric value is used as an index to choose from which operand the value to be entered in the accumulator has to be taken.
The value 0 refers to the first operand. When the accumulator value is larger than the number of operands, the last operand is entered.
Example:

```
LD      destinat
MUX     QuoteX, QuoteY, QuoteZ, -1
```

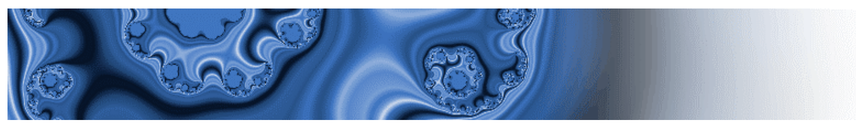
Supposing that the variable destinat is 3, the variable QuoteZ is entered in the accumulator.

## MAX, MIN functions

MAX and MIN enter into the accumulator, the largest or the smallest value between all the operands and the accumulator before the operation.
Example:

```
LD      -400
MIN     PositA, PositB, PositC
```

The smallest value between –400 and PositA, PositB and PositC is entered in the accumulator.



### LIMIT function

LIMIT limits the present accumulator value between the smallest value given by the first operand and the largest value given by the second operand. The result is entered in the accumulator.
Example:

```
LD      current
LIMIT   0, currMax
```

If the current is between 0 and currMax, the accumulator takes the current value. When the current is smaller than 0, the accumulator takes 0. When the current is larger than currMax, the accumulator takes the currMax value.

## 4. GPLC PROJECT

### *DESCRIPTION*

A GPlc project contains all the elements (source modules, memory maps and tasks definition) necessary to create a machine code file (file.COD) to be sent to an AXV driver.
All the necessary information to develop a GPlc project are stored in files with .PPJ extension.
The option "Open project" allows to select a .PPJ file to manage a project.

### *GPLC PROJECT COMPONENTS*

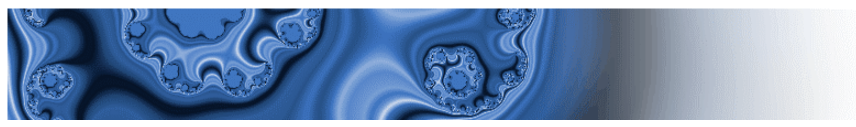
A GPlc is composed of the following elements:

- one or more source modules PLC IEC1131-3;
- one .IMG file including the driver memory map where the created machine code will be stored;
- the link between codified programs in the source modules and the AXV driver executive tasks.

All the above elements are managed with the GPlc dialog-box which is accessible in the option "Project – Settings" of the Menu Bar.

### Source modules

The source modules are made of ASCII files with .PLC extension (not compulsory).
The text must be in accordance with the previous paragraphs and the IEC 1131-3 standard.
The source text can be edited using the integrated text editor or any other ASCII editor.



## IMG Files

IMG files include the description and the values of the driver memory map where the machine code will be stored.
An IMG file is linked one to one to a driver firmware version. This means that a single IMG file exists for each firmware version and vice versa.
A 32 bit code, included in the IMG file and in the firmware, performs the link between them.
The IMG ASCII file must not be modified with editors or similar applications.
Usually, the IMG files linked to the driver firmware versions are included in the PLC project directory.
A missing IMG file can be downloaded from the driver.
The machine code originated by the PLC compiler includes the IMG file identifier code. This way, the driver establishes whether the received code is compatible with its own firmware, if not the PLC execution is disabled.

## TASK

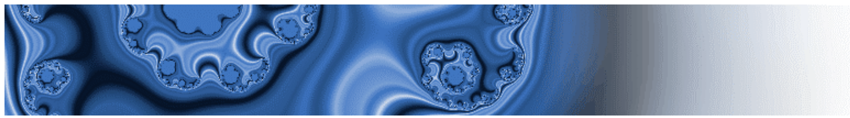
GPlc foresees the link between the drive executive tasks and the programs declared in the source code with the structure PROGRAM .. END_PROGRAM.
This link is carried out with the GPlc dialog-box "Project – Settings".
Presently, the AXV versions provide three tasks with the following settings:

| Name | Period |
|---|---|
| Init | 8 ms |
| Slow | 8 ms |
| Fast | 250 µs |

A PLC project does not require to define the programs to be linked to all tasks. It is possible to develop projects working on a single task among those available.
The task **Init** is called after the driver reset and remains active until its associated program activates the **Slow** and **Fast** tasks by means of firmware variables.
When the task **Init** is not linked to any program, the tasks **Slow** and **Fast** are automatically activated after the driver reset.

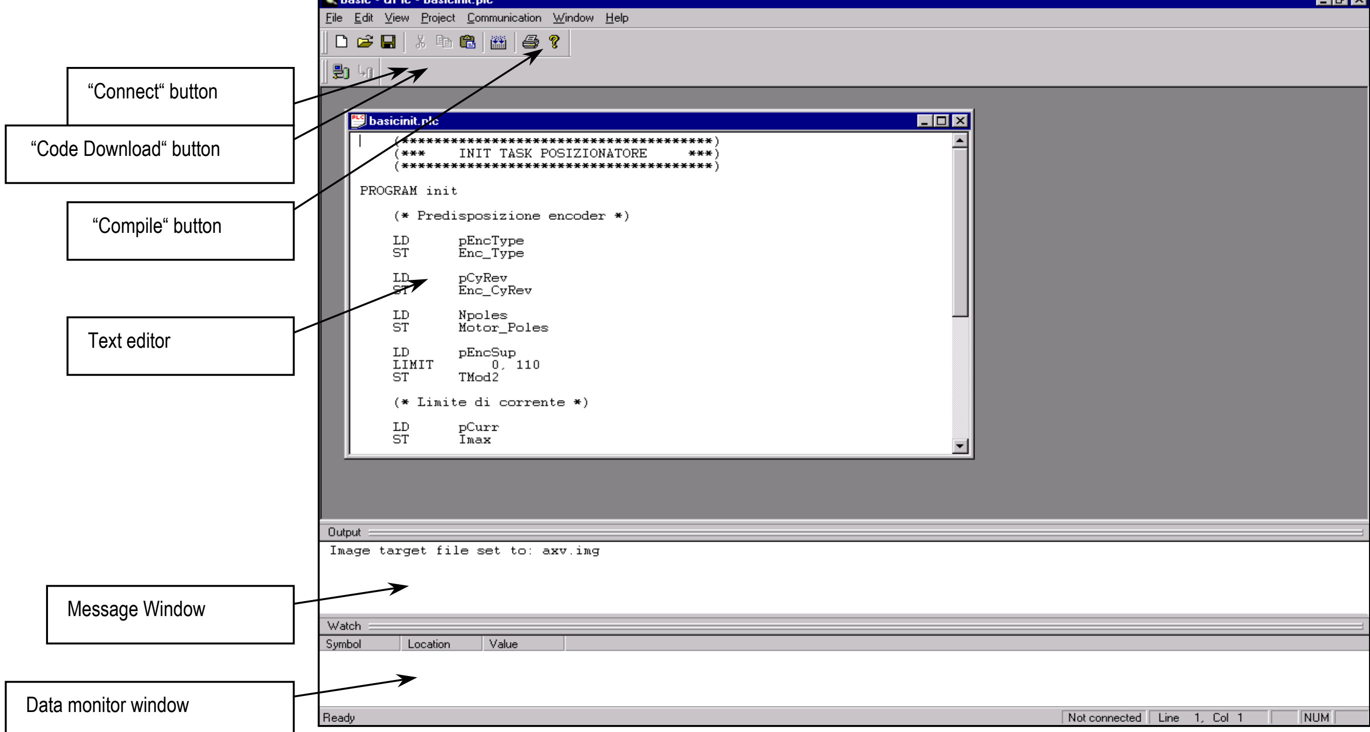
## *CREATING AND RUNNING A GPLC PROJECT*

A new GPlc project is created using the Menu Bar option "File – New project", the dialog-box asks for a project name and the work directory.
The project file .PPJ as well as all the files created by the program are stored in the work directory. The source modules and the file .IMG can be stored in any other location.
To run an existent project, use the option "Open project".



# 5. PROGRAM DESCRIPTION

*FEATURES*



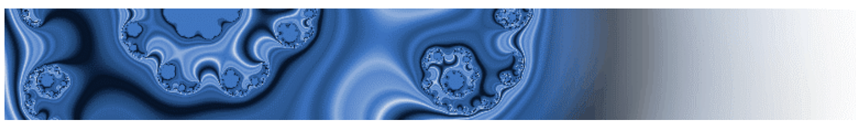

The main GPlc elements are:

- Source module Editor.
- Compiler.
- Communication interface.
- Data monitor.

With the above order, the elements perform the different steps to create a PLC application, the steps are:

- Editing the source code.
- Compiling the source code.
- Sending the originated code to the driver.
- Program debugging by displaying the run-time variables.

An output window is always available to display compiling errors and the GPlc executive messages.



*EDITOR*

The integrated editor features are typical of Windows environment editors and provide with:

- Text selection.
- Cut, copy and paste command.
- Find and replace.
- Drag  and drop selected text.
- Move selected text.

The above commands are accessible from the "Edit" menu, which is activated when at least one text file is open.
The command "File – Open" opens a PLC source file or any other text file.
Moreover , following features are available:

- Row and column number displaying in the status bar.
- Automatic positioning on compiling errors.

To position on the text block with compiling errors, double click the left mouse button on the error line displayed in the "output window" (see paragraph 0).

*COMPILER*

The command "Project – Compile project" starts the compiler to process all the project files one by one and then to originate the machine code using the information in the IMG file.
During this process, the "Output window" displays each process phase and the list of errors and warnings sent out by the compiler while processing.
If there are no errors, the compiler creates a machine code file .COD for the driver.
At the end of the compiling process, the compiler creates a report file (.LST) where all the originated code informations (assembler instructions, variable allocation, memory map etc.) are listed.

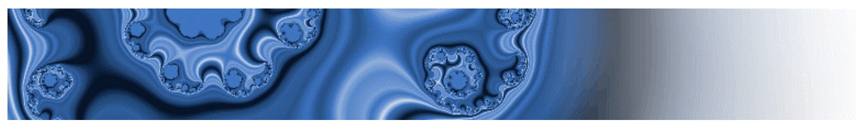
*COMMUNICATION INTERFACE*

The communication interface uses the Slink3 protocol and provides the following features:

- Communication setup
- Download the file .COD into driver.
- Upload the file .IMG from driver.
- Get data for variable monitor.

To download the code into the driver, follow this procedure:

- When not connected, enable the communication interface using the option "Communication – Connect" in the GPlc menu.



- If necessary, use the option "Communication – Settings" to set up the connection parameters. The parameter settings for a serial connection are: 38400 baud, no parity, 8 data bit, 1 stop bit.
- Use "Communication - Download code" to start the code download.

The download status is displayed in the "Output window".
The connection status is displayed in the status bar.
When the driver firmware version is not compatible with the file IMG, the download is disabled (see paragraph 0).
In this case, either select a different IMG file for the project or upload the data memory map into the selected IMG file using the option "Communication – Upload IMG file".

*DATA MONITOR*

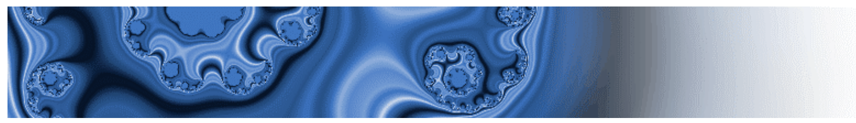
The user can enter in the "Watch window", the program variable names to be displayed during the program execution.
While the connection is enabled, the current variables value is displayed and constantly updated .
To use the data monitor, follow this procedure:

- Compile the current project.
- When not connected, enable the communication interface using the option "Communication – Connect" in the GPlc menu.
- Download the code.
- Use the mouse to point the first free cell in the "Symbol" column of the "Watch window", click the mouse left button to enable the editing and enter the wished variable name.

– or –

- Select a variable name in the editor and use the mouse to drag it to the "Watch window".
- When several variables have the same name, a dialog box will allow to select the wished variable.
- If the name displayed in the "Watch window" doesn't correspond to any variable in the field "Value", "object not found" will appear.
- The "Location" and the "Value" fields display the position where the project variables are used and what value they have.
- When some errors occur, the "Value" field displays the string "..." with an undefined value.

Please note that only the programs declared and used variables are the valid variables of a project. The variables declared but not used are not originated by the compiler and then have no related value.



# 6. PARAMETERS INTERFACE

### *INTRODUCTION*

Inside a GPLC program it is possible to use variables which can be managed by an external configuration or supervision programs (i.e. AXV Cockpit). These variables must be located at defined memory addresses and linked to an index that allow them to be integrated into the system database.
Some variables can keep their value permanently using a configuration command: this particular type of variables are defined Parameters.

### *PARAMETERS and VARIABLES DECLARATION*

It is possible to directly declare data Variables and Parameters using the structure AT.
AXV firmware provides the parameters to data blocks 10, 11, 12,13, 20, 21, 22 and 23.

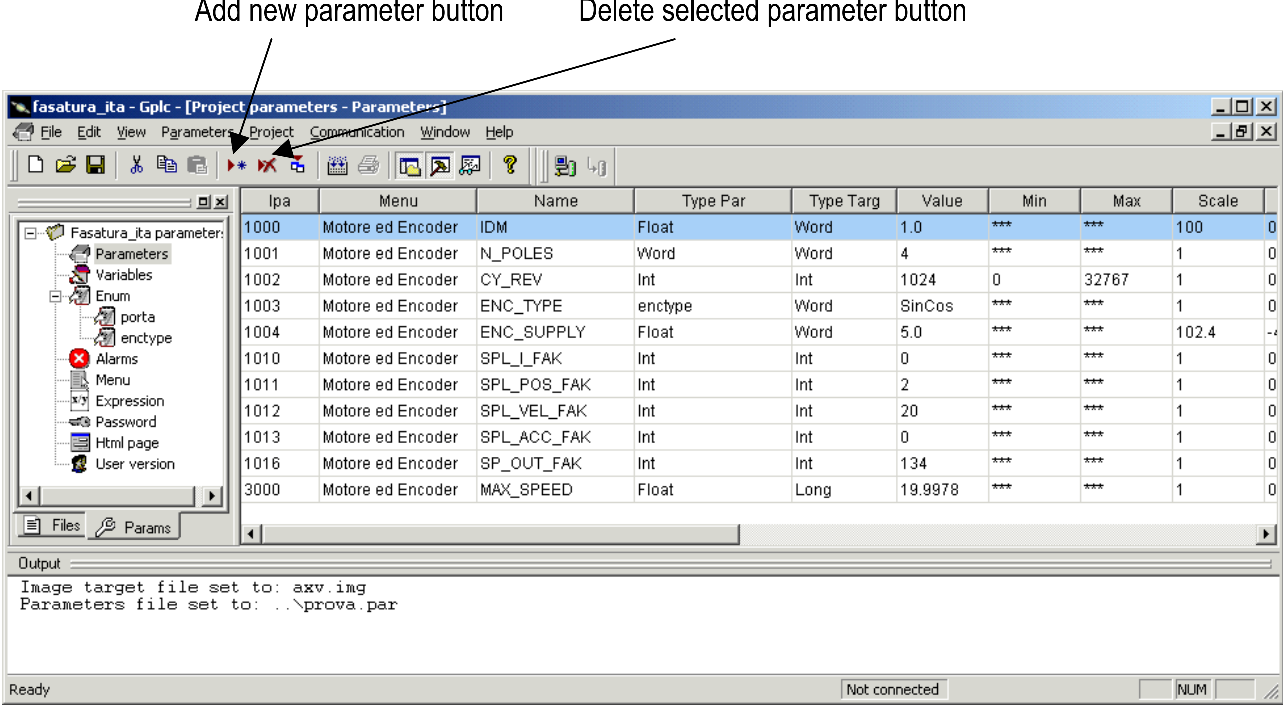
| Data Block | Parameter type | Parameter number | Parameter index (IPA) | Save to FLASH |
|---|---|---|---|---|
| 10 | 16-bit parameters | 1000 | 1000 | YES |
| 11 | 32-bit parameters | 528 | 3000 | YES |
| 12 | Bit parameters | 128 | 5000 | YES |
| 13 | Float parameters | 500 | 4000 | YES |
| 20 | 16-bit parameters | 640 | 7000 | NO |
| 21 | 32-bit parameters | 640 | 9000 | NO |
| 22 | Bit parameters | 128 | 11000 | NO |
| 23 | Float parameters | 128 | 10000 | NO |

For example, in order to define a DINT parameter with the name pMaxSpeed and IPA = 3030, declare:

```
pMaxSpeed AT %MW11.30;
```

But this is not the easiest way to do it. GPLC provides a dedicated window to declare externally accessible Variables and Parameters and automatically generate the corresponding AXV cockpit compatible file to manage them.

Figure below shows the parameters declaration window.




| Ipa | Menu | Name | Type Par | Type Targ | Value | Min | Max | Scale |
|---|---|---|---|---|---|---|---|---|
| 1000 | Motore ed Encoder | IDM | Float | Word | 1.0 | *** | *** | 100 |
| 1001 | Motore ed Encoder | N_POLES | Word | Word | 4 | *** | *** | 1 |
| 1002 | Motore ed Encoder | CY_REV | Int | Int | 1024 | 0 | 32767 | 1 |
| 1003 | Motore ed Encoder | ENC_TYPE | enctype | Word | SinCos | *** | *** | 1 |
| 1004 | Motore ed Encoder | ENC_SUPPLY | Float | Word | 5.0 | *** | *** | 102.4 |
| 1010 | Motore ed Encoder | SPL_I_FAK | Int | Int | 0 | *** | *** | 1 |
| 1011 | Motore ed Encoder | SPL_POS_FAK | Int | Int | 2 | *** | *** | 1 |
| 1012 | Motore ed Encoder | SPL_VEL_FAK | Int | Int | 20 | *** | *** | 1 |
| 1013 | Motore ed Encoder | SPL_ACC_FAK | Int | Int | 0 | *** | *** | 1 |
| 1016 | Motore ed Encoder | SP_OUT_FAK | Int | Int | 134 | *** | *** | 1 |
| 3000 | Motore ed Encoder | MAX_SPEED | Float | Long | 19.9978 | *** | *** | 1 |

Each externally accessible parameter is identified inside the GPLC code with the name:

pParName

Each externally accessible variable is identified inside the GPLC code with the name:

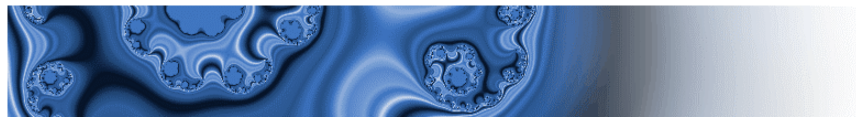
vParName

For example parameter IDM will be used inside the program code as follows:

```
LD      pIDM
ST      Imax
```

To add a new parameter, open the parameters window, press the "Add new parameter" button and select the parameter type (BOOL, INT etc.). The system will assign the first free Index of the corresponding DataBlock for the new parameter. Each parameter is characterized by the following fields :

| | |
|---|---|
| <IPA> | Parameter index. Automatically assigned by the system. |
| <MENU>* | Indicates the AXV Cockpit display menu to which the parameter is associated |
| <NAME> | A mnemonic name used to identify the parameter |
| <PARTYPE>* | Type of the parameter shown in the table (see example below) |
| <VAL>* | Displayed value of the parameter |
| <MIN>* | Minimum value accepted for the parameter |
| <MAX>* | Maximum value accepted for the parameter |

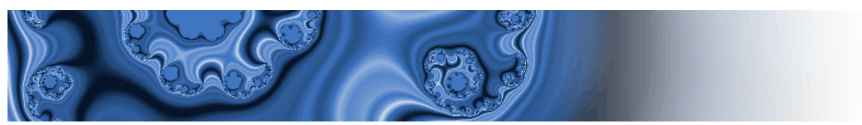


| | |
|---|---|
| <VARTYPE>* | Type of the parameter sent to the drive |
| <SCALE>* | Multiplying factor between displayed value and sent value |
| <OFFS>* | Offset between displayed value and sent value |
| <UNIT>* | Measure unit displayed in the field unit |
| <DESCR>* | Parameter description displayed in the "Description" field |
| <NOTE>* | This field is displayed as footnote of the parameter and can contain additional information (e.g. value range) |

Most of these fields (identified by *) are not related to the parameter itself but only to its management from the configuration program AXV Cockpit and have no effect if the parameter is managed by means of a different system (for example from an industrial panel).

The same consideration is valid for additional functions such as ENUM, MENU and EXPRESSION definition.

For additional information on these fields see the last part of AXV Cockpit user manual where the composition of an AXV Cockpit .par file is described.



# 7. FIRMWARE DRIVE INTERFACE

*INPUT, OUTPUT AND DATA BLOCKS*

Using the structure AT (see paragraph 0) the PLC programs can access and refer to the firmware variables.
The variables of AXV firmware versions supporting the PLC programming are accessible defining INPUT, OUTPUT and DATA BLOCKS areas.
The table shows the prefix to be used in order to define the interface areas.

| Area | Location prefix |
|---|---|
| INPUT | I |
| OUTPUT | Q |
| DATA BLOCK | M |

For example, the declaration:

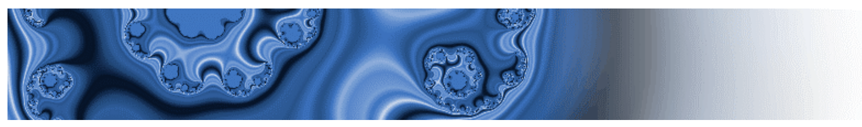
```
Inp4 AT %IX0.4 : BOOL;
```

defines the boolean variable Inp4 in input area, block 0 index 4 (5th AXV digital input).
The declaration:

```
Rg_SpRef AT %MW0.146 : DINT;
```

defines the double integer variable Rg_SpRef in data block 0, word index 146 in data blocks area (speed reference on AXV drive).
The table below shows the areas and blocks available in the present AXV firmware version:

| Area | Block | Description | Number of elements |
|---|---|---|---|
| INPUT | 0 | Digital inputs | 16 |
| OUTPUT | 0 | Digital outputs | 16 |
| INPUT | 1 | Analog inputs | 3 |
| OUTPUT | 1 | Analog outputs | 4 |
| DATA BLOCK | 0 | DSP variables | 641 |
| DATA BLOCK | 1 | Maximum current | 1 |
| DATA BLOCK | 2 | DSP control bits | 16 |



| DATA BLOCK | 3 | DSP error bits | 16 |
|---|---|---|---|
| DATA BLOCK | 4 | DSP control bits | 16 |
| DATA BLOCK | 5 | Task management bits | 3 |
| DATA BLOCK | 10 | 16-bit parameters | 248 |
| DATA BLOCK | 11 | 32-bit parameters | 128 |
| DATA BLOCK | 12 | Bit parameters | 128 |

*FIRMWARE INTERFACE FILES*

In order to save the programmer from declaring all the firmware variables he is going to use, some PLC source files, providing the full set of variables available in AXV firmware versions and linked with IMG files, are supplied. One of these files must be included in any GPlc project and its name is AxvvarsXX.plc where XX is the file release.

Very important: those files should never be mixed with one another or modified to avoid variables misuse ( wrong meaning for variables).

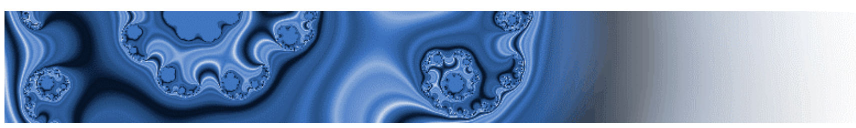
The firmware variables work on "images", they access the code directly only during the program input and output phases, otherwise they access an automatically created local copy. Those phases are scheduled before and after the relative program execution and the "image" variable can change a lot of times during the program execution but only the last value is available to the system.
On the contrary the variables and parameters declared by the user are directly accessible.

The most useful variables provided by the Ax-V family motion controllers are the following:

*DIGITAL I/O*

8 digital input and output on C1 and C2 terminal boards:

| Name | Type | Description | Values |
|---|---|---|---|
| inp0… inp7 | BOOL | Digital inputs | > 20 V = True; < 10 V = False |
| out0…out7 | BOOL | Digital outputs | True = Vcc-2 V; False = 0 V |



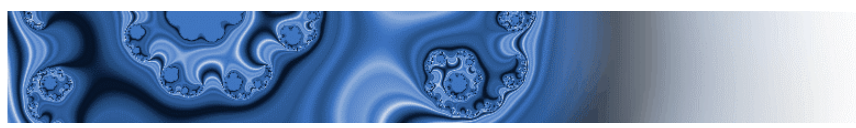
## *ANALOG I/O*

3 differential analog input ±10 V with a A/D 12 bits converter and an internal multiplier *16, 4 analog outputs ±10 V with a D/A 10 bits converter:

| Name | Type | Description | Values |
|---|---|---|---|
| ainp0...ainp2 | INT | Analog inputs | $\pm 10$ V = $\pm 2^{15}$ |
| aout0...aout2 | INT | Analog outputs | $\pm 2^{9}$ = $\pm 10$ V |

## *ENCODER SIGNALS READING*

The brushless servo motor control signals are used mainly for two functions: speed/position loop feedback and three stator currents modulating in order to get a 90 electrical degree phase difference compared to the permanent magnet field.
At the beginning, before the index intervention, these two functions are carried out by two separated sensors often integrated in the same device. In order to keep the stator field in the desired position, it is necessary to derive the absolute position during the electric switching on revolution: usually Sincos absolute sensors ( one sinusoid per revolution ) or Hall effect sensors ( 6 positions resolution for an electric revolution ) are used. The speed/position feedback loop requires the higher possible resolution as it defines the control loop performance. For this reason, a stepper ( not absolute ) track waveform of thousands pulses per revolution is used. This waveform can be digital ( square pattern pulses ) or analog ( sine pattern pulses ); in this last case, the drivers AX-V apply an interpolation within a single pulse enhancing the resolution of $2^{14}$ and providing a very high precision performance for low speed axe blocked applications. By means of a period meter, the digital signals are interpolated too, getting a resolution increment of $2^{14}$, but this interpolation is impossible when the axe is blocked, this is why this solution should not be used when the common use is axe blocked. After the first index intervention, as the index mechanical position is known, the field modulation too is based on the sensor absolute signal with the best resolution.
An automatic phasing routine is foreseen as well, allowing the use of an encoder without absolute tracks; when a type 4 or 7 (see later ) encoder is selected this routine is automatically activated upon the first system switch on and recognizes the electrical position through a vibration ( see paragraph Stepper Encoder Automatic Phasing).



The AX-V motion drivers can simultaneously read the following position sensor signals :

| | |
|---|---|
| Stepper Analog/Digital | suffix AD |
| Absolute Analog (SinCos) | suffix AN |
| Hall sensors | suffix HA |
| Stepper Digital | suffix DI |

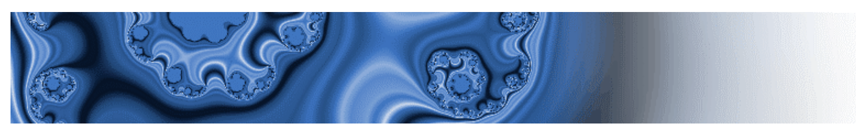
To select the encoder type to be used, set the correct value for the following variables (usually this operation is done only once when the program Init is activated) :

| Name | Type | Description | Values |
|---|---|---|---|
| Enc_Type | INT | Encoder type selection | See note [1] |
| Enc_CyRev | INT | Pulse numbers per revolution for Stepper encoder | |
| Motor_Poles | INT | Motor magnetic pole number | |
| Enc_Port | BOOL | Used port ( only for encoder with stepper digital waveform) | 0 = Port S2; 1 = Port S1 |

1) Encoder types:
- 1 = SinCos 5 tracks (field on AN, position on AD)
- 2 = Digital 6 tracks (field on HA, position on DI)
- 3 = Analog 6 tracks (field on HA, position on AD)
- 4 = Only Digital stepper (field on DI, position on DI)
- 5 = SinCos 2 Absolute Tracks or Resolver (field on AN, position on AN)
- 6 = Only Hall sensors (field on HA, position on HA)
- 7 = Only Analog + index stepper (field on AD, position on AD)
- 8 = SinCos 5 tracks with Digital stepper part (field on AN, position on DI)

The input AD (connector S2, pin 1, 2, 14, 15) can read either an analog or a digital stepper track. The selection is done automatically according to the encoder type and the selected port.

By detecting the above variables, the system interprets the sensor signals and defines the position locations.
The registers xx_ViPu represent the pulse number per revolution with stepper track enhanced resolution (Enc_CyRev * $2^{14}$).
The hardware counter reads the positions and after interpolation these are entered in two registers xx_ViPo and xx_ViTu (xx = relative suffix). xx_ViPo is the position within the current revolution and it is always a positive number between 0 and xx_ViPu.
Whereas xx_ViTu represents the number of revolutions carried out and is a 32 bit signed number.
The speed values are calculated as a difference between two ticks (125 µs) and entered in xx_PeSp registers.

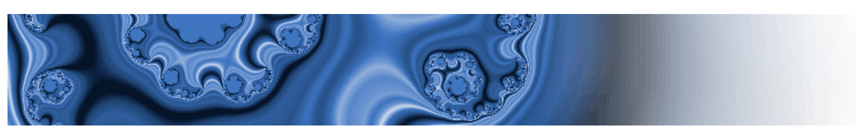


Variables:

| Name | Type | Description | Values |
|---|---|---|---|
| Ad_ViPo | UDINT | Position on the revolution track AD | 0 … Enc_CyRev * $2^{14}$ per 1 revolution |
| Ad_ViTu | DINT | Revolutions number track AD | $\pm 2^{31}$ |
| Ad_ViPu | UDINT | Pulse number per revolution with resolution enhancement | Enc_CyRev * $2^{14}$ |
| Ad_PeSp | DINT | Speed AD with resolution enhancement | |
| An_ViPo | UINT | Position on the revolution track AN | 0 … $2^{14}$ per 1 revolution |
| An_ViTu | DINT | Revolutions number track AN | $\pm 2^{31}$ |
| An_PeSp | INT | Speed AN with resolution enhancement | |
| Di_ViPo | UDINT | Position on the revolution track DI | 0 … Enc_CyRev * $2^{14}$ per 1 revolution |
| Di_ViTu | DINT | Revolutions number track DI | $\pm 2^{31}$ |
| Di_ViPu | UDINT | Pulse number per revolution with resolution enhancement | Enc_CyRev * $2^{14}$ |
| Di_PeSp | DINT | Speed DI with resolution enhancement | |
| Ha_ViPo | UINT | Position on the revolution track HA | 0 … 24575 per 1 revolution |
| Ha_ViTu | DINT | Revolutions number track HA | $\pm 2^{31}$ |
| Ha_PeSp | INT | Speed HA with resolution enhancement | |

Variables useful to read and use the index signals:

| Name | Type | Description | Value |
|---|---|---|---|
| First_AdIndex | BOOL | First index indicator AD crossed after the system switch on | 1 = crossed; 0 = Not crossed |
| First_DiIndex | BOOL | First index indicator DI crossed after the system switch on | 1 = crossed; 0 = Not crossed |
| Ad_IndexOk | BOOL | The system rises this bit when an AD index is crossed and it remains high for one tick of task Slow (8 ms) | 1 = crossed; 0 = Not crossed |
| Di_IndexOk | BOOL | The system rises this bit when an DI index is crossed and it remains high for one tick of task Slow (8 ms) | 1 = crossed; 0 = Not crossed |
| Iad_ViPo | DINT | Position in the AD revolution where the last index acquired is crossed | Enc_CyRev * $2^{14}$ = 1 revolution |
| Iad_ViTu | DINT | Revolution AD where the last index acquired is crossed | $\pm 2^{31}$ |
| Idi_ViPo | DINT | Position in the DI revolution where the last index acquired is crossed | Enc_CyRev * $2^{14}$ = 1 revolution |



## *INCREMENTAL ENCODER AUTOMATING PHASING ROUTINE*

The system firmware integrates a routine able to use the encoder with the stepper track only (encoder type 4 and 7). Due to a lack of absolute position sensors, when the system is switched on, a rough detection of the electrical position (through vibration) until the first index crossing is necessary.
This procedure is called "Automatic Phasing" and can be started setting to 1 the bit StartFas. When the phasing is ended the bit FasatOk is risen. During this operation (about 2 sec.) a sequence of current pulses of increasing amplitude comprised between 0 and Imax is impressed in motor phases. The maximum current value must be written in variable Imax before starting this procedure.
This procedure is normally started from Init task and the Fast and Slow tasks are started after the procedure end.

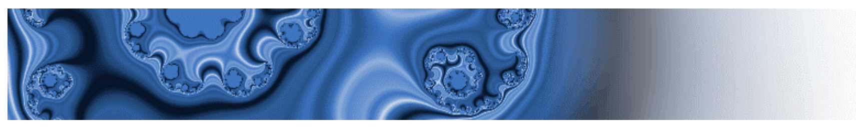
Example (in program Init):

```
LD      200
ST      Imax        (*set 2 Arms as phasing current*)
LD      inp0                (*wait enabling in inp0*)
ST      StartFas    (*after first enabling, the phasing starts*)
LD      FasatOk     (*wait phasing end*)
ST      stFastTsk           (*start program Fast*)
ST      stSlowTsk           (*start program Slow*)
```

| Name | Type | Description | Values |
|---|---|---|---|
| StartFas | BOOL | Start automatic phasing procedure | |
| fasatOk | BOOL | Phasing ended | |

## *ENCODER SIGNALS REPETITION*

The encoder signal used for speed/position feedback loop can be repeated with the desired ratio on connector S1. This connector can be setup either as input or output; to enable signal repetition, set bit abrenc = 1.
The ratio is settled by means of two variables (multiplication and division coefficient). The repetition frequency limit is 500 kHz, when this frequency is exceeded the drive puts alarm on, as some counts could be lost. Modifying the variable Se_SpMax value enables the threshold alarm decreasing in order to protect external reading device with limited pass band.
The index can be repeated with a desired impulses step limited to $2^{30}$. After enabling index repetition, the programmer can set the position of the first repeated index with reference to the first master index. Note: The next index will be repeated with the frequency value SiStep independently from the master index.



Example:

```
LD      inp1            (*read digital input 1*)
AND     First_AdIndex   (*verify that a master index is crossed*)
ST      abrindex        (*if 1 enable index repetition*)
LD      CntNLt1         (*Last index position*)
ADD     1000            (*add 250 (1000/4) repeated pulse*)
ST      SiFirstIndex    (*set the first index position*)
```

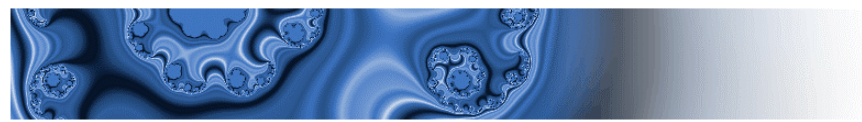
The first index is repeated after 250 simulated encoder pulses following the master index.

| Name | Type | Descriprion | Values |
|---|---|---|---|
| Abrenc | BOOL | Enable encoder repetition.<br>A jumper over pin 13 and 23 of S2 connector is necessary. | 0 = disable<br>1 = enable |
| Se_MulFak | UINT | Multiplication coefficient for encoder repetition | $1 \ldots 2^{16}-1$ |
| Se_DivFak | UINT | Division coefficient for encoder repetition | $1 \ldots 2^{16}-1$ |
| Se_SpMax | UDINT | Speed limit for encoder repetition. | f[Hz] * 2^11 |
| Abrindex | BOOL | Enable index repetition (encoder repetition must be enabled) | 0 = disable<br>1 = enable |
| SiStep | DUINT | Index repetition step | Desired step of repeated pulses * 4 |
| SiFirstIndex | DUINT | Position where the first index is to be repeated | Value of encoder count * 4 |
| CntNlt1 | DUINT | Position where the last master index is crossed | Value of encoder count * 4 |

### *CURRENT LOOP*

This loop is the speediest control feature , and has a sampling rate of 16 kHz. There are two current loops executed simultaneously; indeed both direct and quadrature currents components are calculated from the current phase read by the AD converters, and both are controlled to get the desired operation. The quadrature current contributes to the motor torque, while the direct current is usually set to zero (Ic_IsdRef = 0).
Set Spl_Spl = 0 to enable the current ( or torque) control. In this case, the motor torque will be proportional to the quadrature current entered in Cic_IsqRef register.
On the contrary, while working with the speed/location loop, the current loop becomes part of it and its reference is the output of the PID regulator.

The Imax register allows the definition of a symmetrical limit on the reference current which is intrinsically limited to the motor driver rated current.
The current measuring unit is independent from the driver size and is
1 Arms = 100
The current loop gains do not have to be entered or modified by the PLC program as they are system parameters and they are managed with Ax-V Cockpit application (system table).



| Name | Type | Description | Values |
|---|---|---|---|
| Spl_Spl | BOOL | Close the speed loop | 0 = Current; 1 = Speed |
| Cic_IsqRef | INT | Current loop reference | 100 = 1 Arms |
| Imax | INT | Current limit | 100 = 1 Arms |

*POSITION/VELOCITY LOOP*

The Ax-V motion control platform includes a velocity/position control loop that can be closed using a Spl_Spl bit. If this bit is 0, the loop is open and the system works with a current control ( see current loop).
The position loop is managed by DSP with a sampling rate of 8 kHz.
As mentioned, selecting the encoder type defines the position sensor to be used as feedback for the position loop and moreover registers (xx_ViPo, xx_ViTu, xx_PeSp) are copied in Spl_ViPo, Spl_ViTu and Spl_PeSp registers.
Using the bit Rg_PosLoop enables to set the system as a velocity controller or as location controller: in the first case, the reference must be entered in Rg_SpRef. The measurement unit for this parameter ( and for speed in general ) are encoder interpolated counts *( $*2^{14}$) in a loop tick (125 µs).* For example, to assign a motor a reference of 1000 rpm with the encoder of 1024 imp./revolution, set:

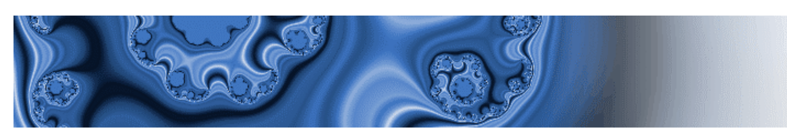
$$Rg_SpRef = 1000/60*1024*2^{14}*125*10^{-6} = 34953$$

When executing a location control, the " patterns generator " processes the stop position target settled in Rg_TurStop and Rg_PosStop and provides the ramp generator with a target speed to reach the required location. When the required position is reached, the bit Rg_PosOk is risen until a new positioning is requested. The motor position error can be defined using the Spl_PosErr variable.

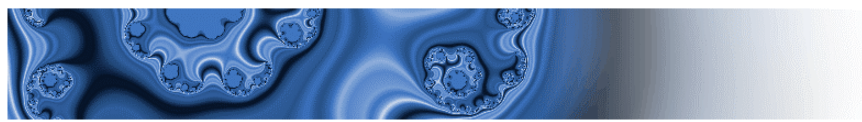
In both cases: the positive (clockwise) and negative (anti-clockwise) speed limits are determined by the Rg_PosspLim and Rg_NegspLim registers and the speed ramp profiles have the slope parameters stored in CwAcc, CcwAcc, CwDec and CcwDec. The relation between those parameters and the real acceleration rad/$s^2$ depends on the encoder pulse number (see next table). Longer ramp pattern can be achieved changing the value of the Rg_ExpRamp variable. This value must be different from 0 (until slower ramp are not required, it is advisable to use the value 1).
It is possible to disable the ramp patterns setting to 1 the bit Rg_RampOff.

The velocity/position gains registers are reported in the following table:

| Name | Type | Description | Values |
|---|---|---|---|
| Spl_Spl | BOOL | Speed loop closing | 0 = Current; 1 = Speed |
| Rg_PosLoop | BOOL | Position loop closing | 0 = Speed 1 = Position<br>(only if Spl_Spl = 1) |



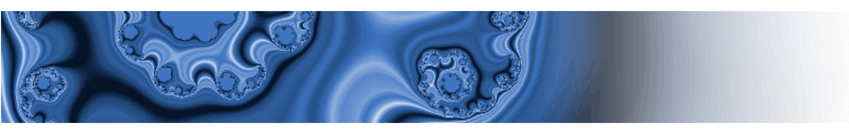
| Name | Type | Description | Values |
|---|---|---|---|
| CwAcc | UINT | Clockwise acceleration | 1 = (24543.7/Enc_CyRev * Rg_ExpRamp) rad/$s^2$ |
| CcwAcc | UINT | Anti-clockwise acceleration | 1 = (24543.7/Enc_CyRev * Rg_ExpRamp) rad/$s^2$ |
| CwDec | UINT | Clockwise deceleration | 1 = (24543.7/Enc_CyRev * Rg_ExpRamp) rad/$s^2$ |
| CcwDcc | UINT | Anti-clockwise deceleration | 1 = (24543.7/Enc_CyRev * Rg_ExpRamp) rad/$s^2$ |
| Rg_ExpRamp | UINT | Ramp slow down coefficient | |
| Rg_RampOff | BOOL | Ramp deactivation | 0 = Enable; 1 = Disable |
| Spl_Spl | BOOL | Speed loop closing | 0 = Current; 1 = Speed |
| Spl_IntFak | UINT | Position integral gain | |
| Spl_PosFak | UINT | Position proportional gain | |
| Spl_VelFak | UINT | Speed proportional gain | |
| Spl_AccFak | UINT | Acceleration proportional gain | |
| Rg_SpRef | DINT | Speed loop reference | Imp. Encoder * $2^{14}$ in 125 µs |
| Rg_PosspLim | UDINT | Clockwise speed limit | Imp. Encoder * $2^{14}$ in 125 µs |
| Rg_NegspLim | UDINT | Anti-clockwise speed limit | Imp. Encoder * $2^{14}$ in 125 µs |
| Rg_PosStop | UDINT | Stop position reference for positioning loop | Imp. Encoder * $2^{14}$ |
| Rg_TurStop | UDINT | Stop revolutions reference for positioning loop | Revolution number |
| Rg_PosOk | BOOL | Reached position bit | 1 = Reached position |
| Spl_PosErr | DINT | Positioning error | Imp. Encoder * $2^{14}$ |



### *DSP CONTROL BIT*

The GPLC application controls some bits that enable and disable the PWM modulation which is usually connected to a digital input.
The bit stFastTsk activates the program Fast (4 kHz) and the bit stSlowTsk activates the program Slow(125 Hz). The driver switch on automatically activates the program Init (125 Hz) until the program Slow replacing it, is not enabled.

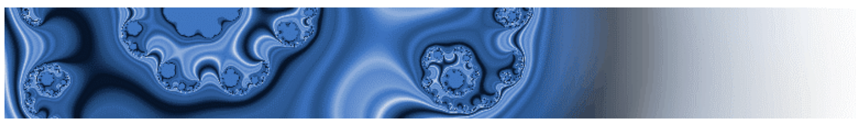
| Name | Type | Description | Values |
|---|---|---|---|
| EnableDrive | BOOL | Enabling drive | 1 = Enable; 0 = Disable |
| stFastTsk | BOOL | Program Fast (125 µs) starting | 1 = Start |
| stSlowTsk | BOOL | Program Slow (8 ms) starting (Init program is activated until Slow is not activated) | 1 = Start |



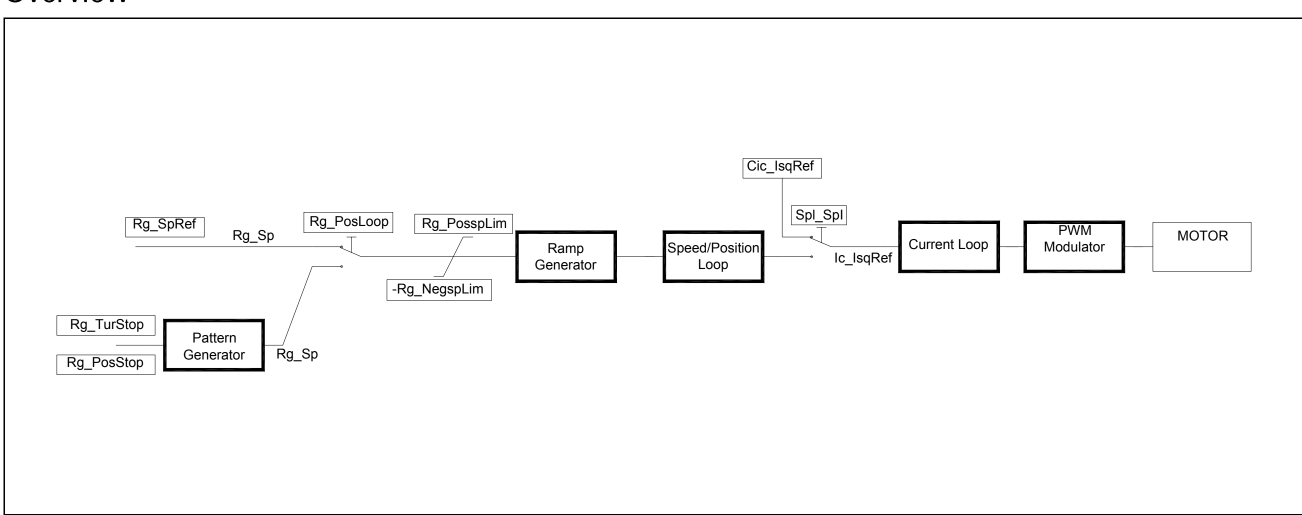
*BLOCK DIAGRAM*

## Symbols

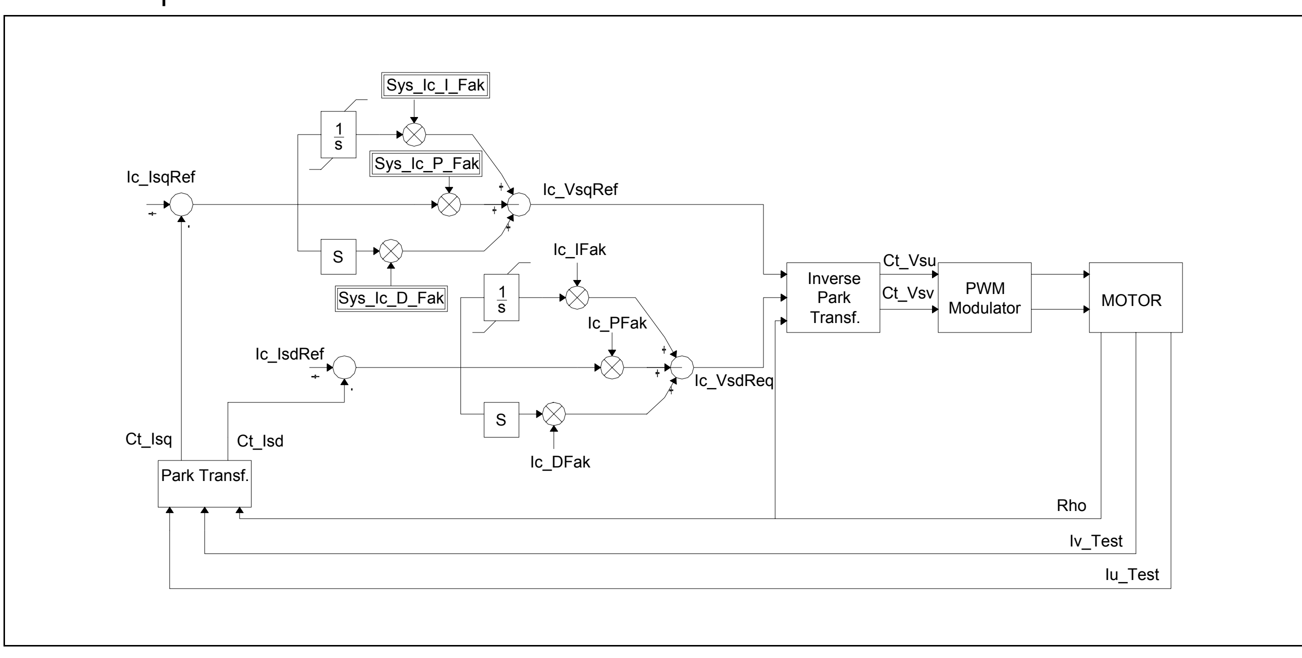
| Symbol | Description |
|---|---|
| varX | System variable |
| varY | GPLC user variable |
| varZ | Variable to be changed with Ax-V Cockpit (system table) |
| | Multiplier |
| Selection Var. | Selector: in this position Var. Selection = 0 (The variable selection can also be a logical expression) |
| + + | Adder |
| SUBASSEMBLY | Subassembly |

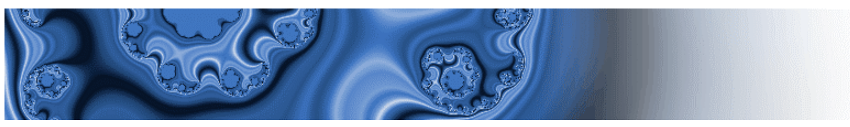


## Overview

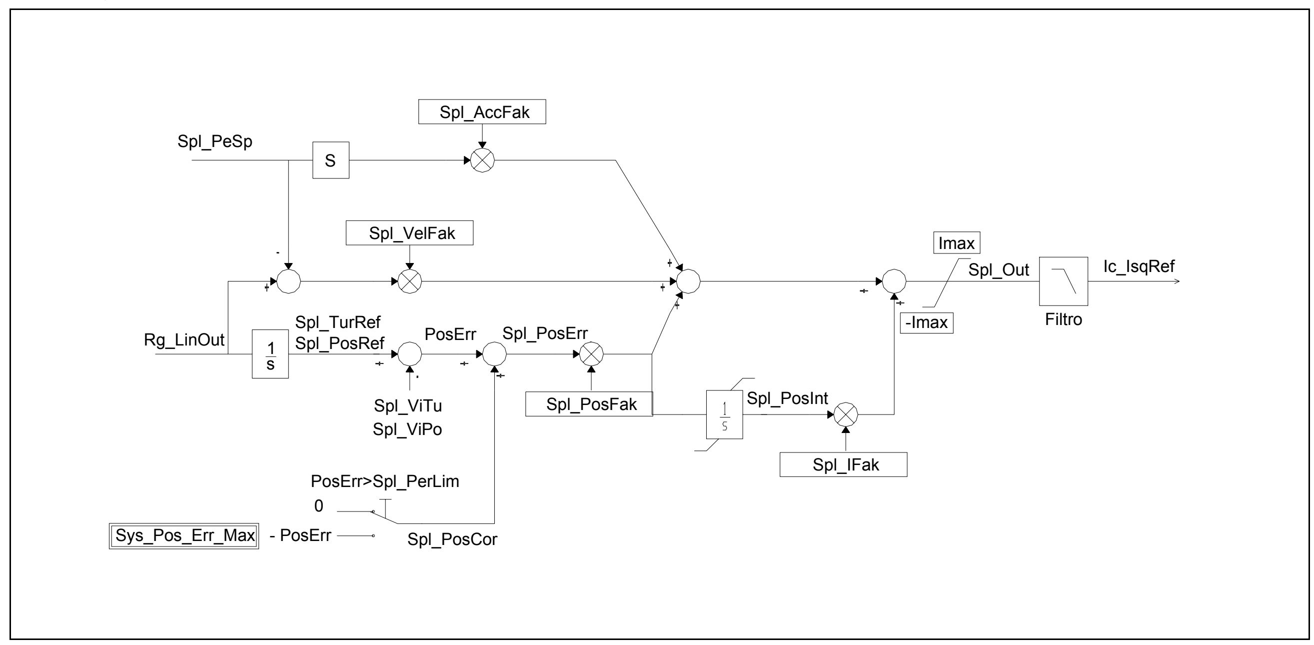



## Current loop

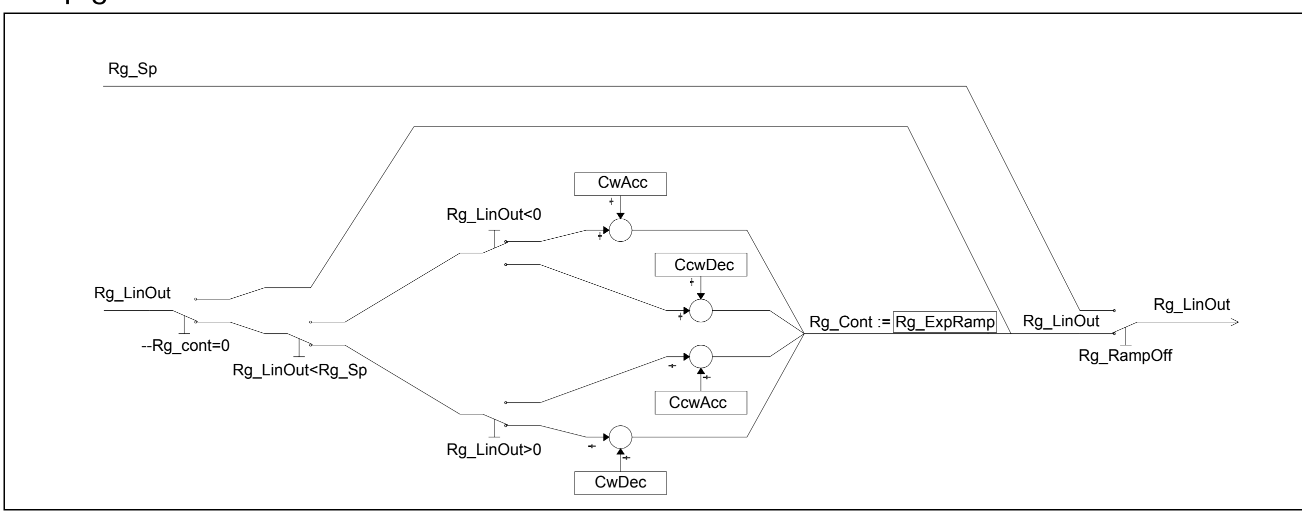


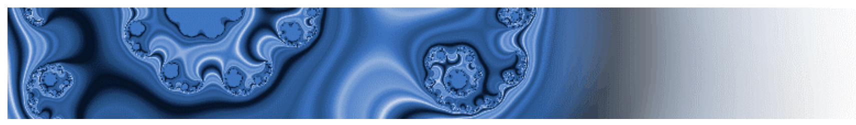


## Velocity/position loop




## Ramp generator





# 8. APPLICATION EXAMPLE

*DESCRIPTION*

In order to help the user to create the first application GPlc and become familiar with this manual, a simple application is explained.

The application to be developed in speed control takes the reference, alternatively from the analog input 0 or from a parameter depending on the digital input 1 value. The enabling command is carried out by the digital input 0.

Moreover, the possibility to change the ramp slope parameters and the loop gain through an Ax-V Cockpit table are foreseen.

The first step is to create the source files: one for each executive tasks and one with the common variable definition.

*CREATING A PROJECT*

Create a folder for the project . Run GPLC and create three new files (File->New) and save them in the newly created folder with the following names:

basicinit.plc
basicslow.plc
basicfast.plc

Copy to the project folder also the AxvVarsXX.plc and file from any of the existing projects in the Phase Motion Control\Apps folder.

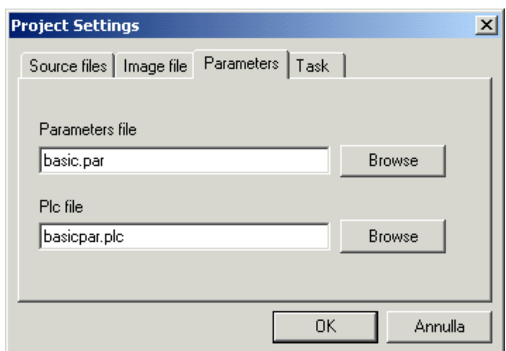
Select "File – New Project" and choose the destination directory and the project (basic) name.

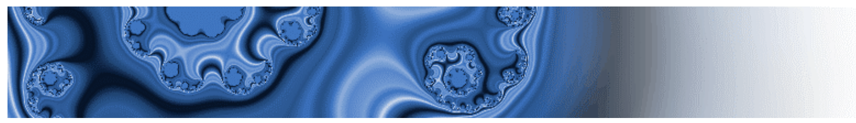
Select Project-> Settings

In the "Source Files" the dialog box add one by one the files you created and the axvvarsXX.plc file:

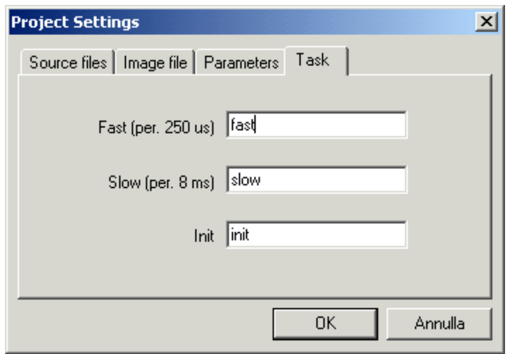
In the Image file dialog box set the name for the AXV memory image file: axv.img

In the parameters dialog box select the name for the AXV Cockpit file the system will create with your application parameters (basic.par) and the internal file the system will create to declare the parameters variables (basicpar.plc).



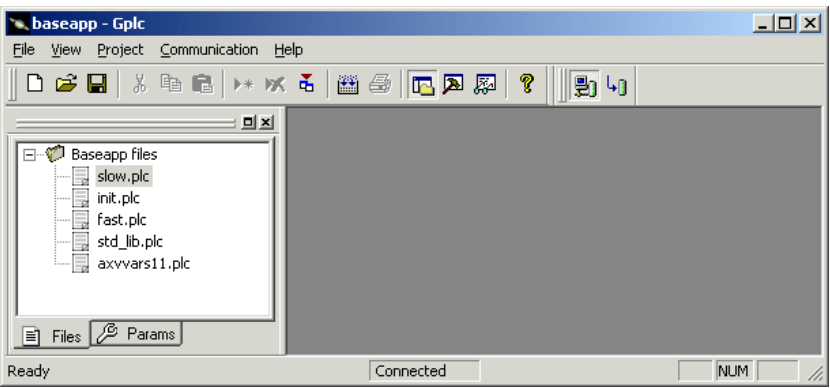


Finally in the Task dialog box insert the names you will use for the programs corresponding to the three tasks (see task paragraph of this manual)




Press OK to complete the project creation.

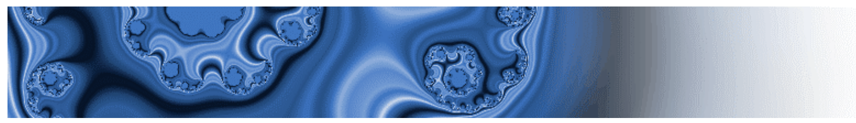
The files window will be filled with the files you added to the project.




Now the project is created and ready to be compiled.
It is possible to open the files and complete the editing without creating the project again.

If the message "Invalid Memory Image File" appears while compiling, it means that either the file axv.img in the directory or the filename chosen in the project settings is missing. To download it from the driver : connect to the driver selecting the option "Communication – Connect" (the driver must be connected and the 24 V signaling lamp switched on), then select " Communication – Upload Image File". If any error occurs, check the connection: the interface 232-485 (the 2 dip switches must be set ON) and the communication settings ("Communication – Settings" – see paragraph Communication Interface).

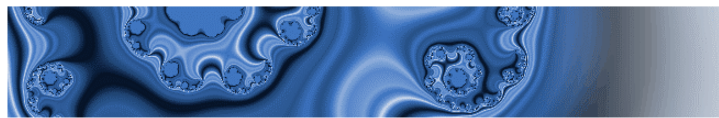
When the project is compiled without errors, it can be sent to the driver using the button "Code Download".



*INIT PROGRAM (file basicinit.plc)*

The variables necessary for the system operation are initialized in this file.
Open the basicinit.plc  file and insert the following code lines

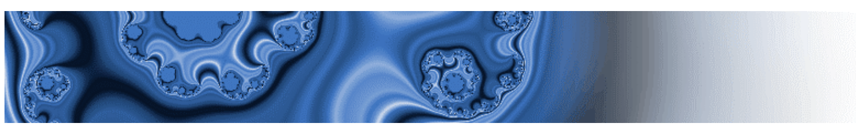
| Code | Comment |
|---|---|
| (***************************************)<br>(***      INIT TASK POSITION CONTROL               ***)<br>(***************************************) | |
| PROGRAM init | |
| (* encoder set up *) | |
| LD pEncType<br>ST EncType | Take the encoder type from pEncType parameter |
| LD pCyRev<br>ST EncCyRev | Take the pulses number per revolution from pCyRev |
| LD pNPoles<br>ST MotorPoles | Take the motor poles number from Npoles |
| LD pEncSup<br>LIMIT 0, 110<br>ST Tmod2 | Take the pencoder supply voltage from pEncSup |
| (* Current limit *) | |
| LD pCurr<br>ST Imax | Take the current limit from pCurr |
| (* Speed loop *) | |
| LD TRUE<br>ST Spl_Spl | Speed loop set up… |
| LD FALSE<br>ST Rg_PosLoop | … and no location |
| (*      Operative task beginning            *) | |
| LD TRUE<br>ST stFastTsk<br>ST stSlowTsk | After variables set up, begin the executive tasks. |
| END_PROGRAM | |



*SLOW PROGRAM (basicslow.plc)*

This program sets the variables that can be dynamically changed by Ax-V Cockpit.
Open the basicslow.plc file and insert the following code

| Code | Comment |
|---|---|
| PROGRAM slow<br><br>VAR<br><br>InpAbil : BOOL;<br>OutAbil : BOOL;<br><br>END_VAR | Define two local variables |
| (* Manage inputs *)<br><br>LD inp0<br>ST inpAbil | Verify the user enabling command on digital input 0 |
| (* Ramps *)<br><br>LD pCwAcc<br>ST CwAcc<br>LD pCcwAcc<br>ST CcwAcc<br><br>LD pCwDec<br>ST CwDec<br>LD pCcwDec<br>ST CcwDec<br><br>LD 1<br>ST Rg_ExpRamp | Read ramps slopes from parameters |
| (* Gains *)<br><br>LD pKInt<br>ST Spl_IntFak<br>LD pKPos<br>ST Spl_PosFak<br>LD pKSpd<br>ST Spl_VelFak<br>LD pKAcc<br>ST Spl_AccFak | Read loop gains from parameters |
| (* Speed limits *)<br><br>LD 209000<br>ST Rg_PosspLim<br>ST Rg_NegspLim | Set clockwise and anti-clockwise speed limits. |
| (* Enable drive *)<br><br>LD inpAbil<br>ST EnableDrive<br>ST outAbil | Driver enabling |

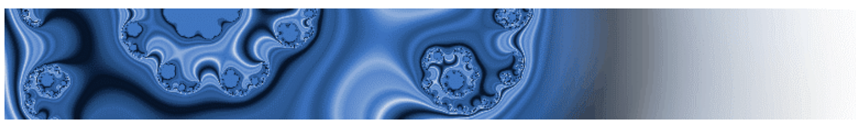


| | |
|---|---|
| (* Manage outputs*)<br><br>LD outAbil<br>ST out0<br><br>END_PROGRAM | If enabled, turn on the digital output 0 |

*FAST PROGRAM (basicfast.plc)*

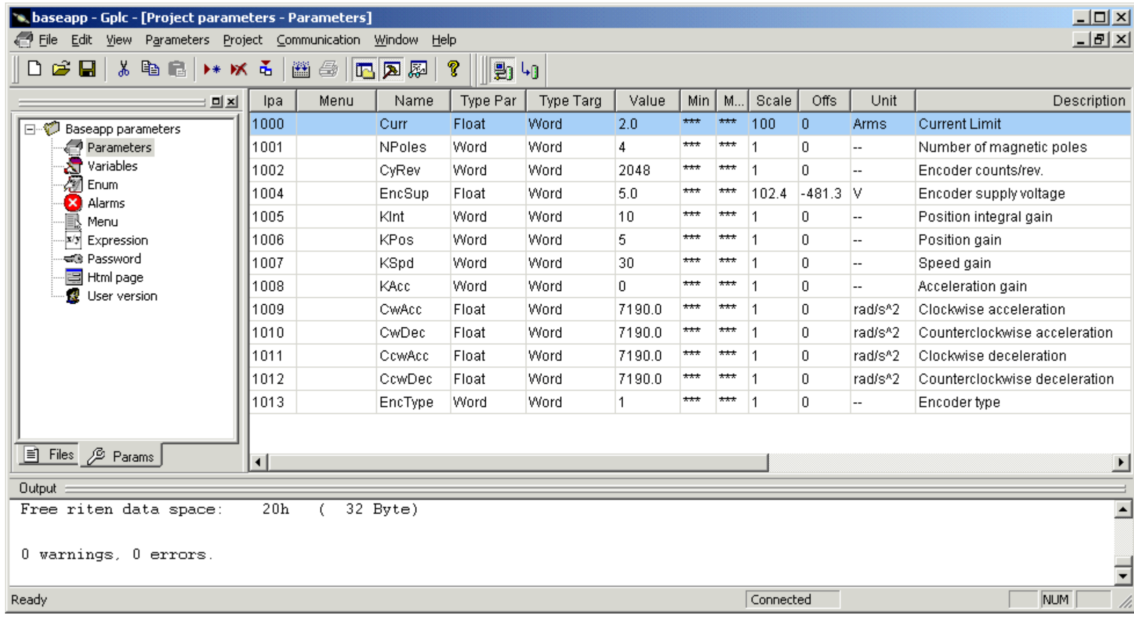
This program is used to read the high frequency speed reference from the analogue input 0.
Open the basicsfast.plc file and insert the following code

| | |
|---|---|
| PROGRAM fast<br><br>VAR<br><br>RifAna : BOOL;<br>Temp32 : DINT;<br><br>END_VAR | <br><br><br><br>Local variable definition |
| (* Manage inputs *)<br><br>LD inp1<br>ST RifAna | <br><br>Verify the reference selection from analog input or parameter |
| LD RifAna<br>JMPCN rifparam<br>LD ainp0<br>ST temp32<br>MUL 16<br>DIV 5<br>ST Rg_SpRef<br>JMP refok | If no reference jump to rifparam<br><br>Read input ainp0<br>ainp0 extention to 32 bit for compatibility with Rg_SpRef<br>Scale span adjustment to have about 3000 rpm at 10 V reference.<br>Write speed reference<br>End reference |
| rifparam:<br>LD pSpRef<br>ST Rg_SpRef<br><br>refok:<br><br>END_PROGRAM | <br>Read parameter SpRef<br>Write reference |



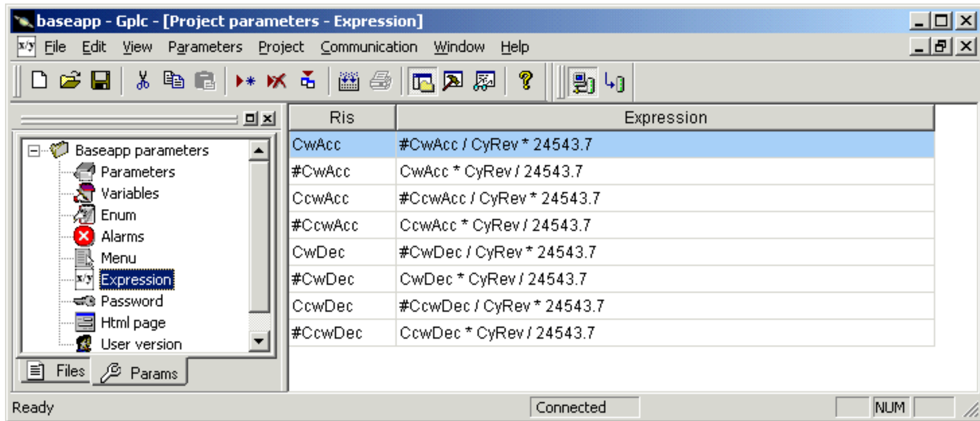
*VARIABLE AND PARAMETERS (basicpar.plc)*

Declare the parameters as shown in figure below:



| Ipa | Menu | Name | Type Par | Type Targ | Value | Min | M... | Scale | Offs | Unit | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | | Curr | Float | Word | 2.0 | *** | *** | 100 | 0 | Arms | Current Limit |
| 1001 | | NPoles | Word | Word | 4 | *** | *** | 1 | 0 | -- | Number of magnetic poles |
| 1002 | | CyRev | Word | Word | 2048 | *** | *** | 1 | 0 | -- | Encoder counts/rev. |
| 1004 | | EncSup | Float | Word | 5.0 | *** | *** | 102.4 | -481.3 | V | Encoder supply voltage |
| 1005 | | KInt | Word | Word | 10 | *** | *** | 1 | 0 | -- | Position integral gain |
| 1006 | | KPos | Word | Word | 5 | *** | *** | 1 | 0 | -- | Position gain |
| 1007 | | KSpd | Word | Word | 30 | *** | *** | 1 | 0 | -- | Speed gain |
| 1008 | | KAcc | Word | Word | 0 | *** | *** | 1 | 0 | -- | Acceleration gain |
| 1009 | | CwAcc | Float | Word | 7190.0 | *** | *** | 1 | 0 | rad/s^2 | Clockwise acceleration |
| 1010 | | CwDec | Float | Word | 7190.0 | *** | *** | 1 | 0 | rad/s^2 | Counterclockwise acceleration |
| 1011 | | CcwAcc | Float | Word | 7190.0 | *** | *** | 1 | 0 | rad/s^2 | Clockwise deceleration |
| 1012 | | CcwDec | Float | Word | 7190.0 | *** | *** | 1 | 0 | rad/s^2 | Counterclockwise deceleration |
| 1013 | | EncType | Word | Word | 1 | *** | *** | 1 | 0 | -- | Encoder type |

Define the expressions for parameters CwAcc, CcwAcc, CwDec, CcwDec as shown in below:



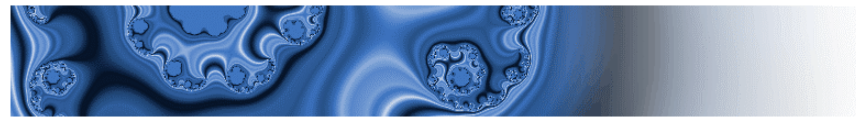
| Ris | Expression |
|---|---|
| CwAcc | #CwAcc / CyRev * 24543.7 |
| #CwAcc | CwAcc * CyRev / 24543.7 |
| CcwAcc | #CcwAcc / CyRev * 24543.7 |
| #CcwAcc | CcwAcc * CyRev / 24543.7 |
| CwDec | #CwDec / CyRev * 24543.7 |
| #CwDec | CwDec * CyRev / 24543.7 |
| CcwDec | #CcwDec / CyRev * 24543.7 |
| #CcwDec | CcwDec * CyRev / 24543.7 |

*AX-V COCKPIT PARAMETERS TABLE*

As mentioned in the example, a lot of variables are parameters to be configured using Ax-V Cockpit. The corresponding AXV Cockpit parameters table is created automatically when the project is compiled and will have the name selected in the Project properties.

***COMPILE THE PROJECT***

Switch on the 24 aux. supply voltage to the drive and connect the serial cable from the PC. In the Communication menu select Connect and then Upload Img file. If the



connection is good GPLC will load the memory map file from the drive and you will see in the output window the message:

```
Loading memory image from target .. completed.
File axv.img updated.
```

If the following error message appears:

```
Loading memory image from target .. failed.
```

verify the communication settings (see communication paragraph of this manual).

Once the memory image file is updated you can compile the project selecting Project->Compile Project

If GPLC can compile the project without errors you can download the application to the drive selecting
Communication-> Download Code.

At the end of download operation the drive will be automatically reaset and begin to execute the new program.