PCSuite IDE+

User Manual

agito

www.agito-akribis.com

Member of Akribis Systems group

Revision History

Version	Description	Date
1.1	Updated Chapter 4.5	7 July 2026
1.0	Initial release	29 July 2024

Contact Information

Manufacturer	Agito Akribis Systems Ltd., Member of Akribis Systems Group
Address	56 Serangoon North Avenue 4 Singapore 555851
Email	info@akribis-sys.com
Website	www.agito-akribis.com

Copyright Notice

©2026 Agito Akribis Systems Ltd.

All rights reserved. This work may not be edited in any form or by any means without written permission of Agito Akribis Systems Ltd.

Products Rights

AGDx, AGCx, AGMx, AGAx, AGIOx, and AGLx are products designed by Agito Akribis Systems Ltd. in Israel. Sales of the products are licensed to Akribis Systems Pte Ltd. under intercompany license agreement.

Agito Akribis Systems Ltd. has full rights to distribute above products worldwide.

Disclaimer

This document was accurate and reliable at the time of its release.

Agito Akribis Systems Ltd. reserves the right to change the specifications of the product described in this document without notice at any time.

Trademarks

Agito PCSuite is a trademark of Agito Akribis Systems Ltd..

Contents

1 Introduction 4
1.1 About this Manual 4
1.2 Introduction to PCSuite IDE Programming Environment 4
2 Getting to Know PCSuite IDE Programming Environment 5
2.1 IDE Programming Environment Interface 5
2.2 Program Execution Steps 7
2.3 Saving User-Program to Controller Flash 7
3 User Program Basic Syntax 8
3.1 User Program Files 8
3.2 Compiler Directives 8
3.3 Constants and Variables 10
3.4 Array 11
3.5 Pointer 12
3.6 Conditional Statements 14
3.7 Loop Statements 15
3.8 Operators 15
3.9 Functions 17
3.10 Tasks and Threads 18
4 User Program Examples 20
4.1 PTP Motion 20
4.2 Jog Motion 20
4.3 Sending Command Commutation 21
4.4 Homing 22
4.5 Digital I/O Handling 23
4.6 Error Compensation Laser Program 24
4.7 Multi-threading Calls 24
4.8 Gear Mode Gear Ratio Switching 24
4.9 Event Functions 27
4.10 Sine Wave Motion 28
4.11 Typical Force Control 28
5 Introduction to Common Keywords 30
5.1 Commutation Related 30
5.2 Homing Related 30
5.3 Motion Related 32
5.4 Motor Parameters Related 43
5.5 I/O Related 45
5.6 System Related 49

1 Introduction

1.1 About this Manual

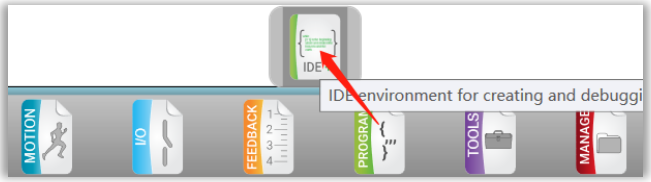
This manual mainly introduces the usage and precautions of the PCSuite IDE programming environment to help users quickly master IDE+ for developing user programs.

1.2 Introduction to PCSuite IDE Programming Environment

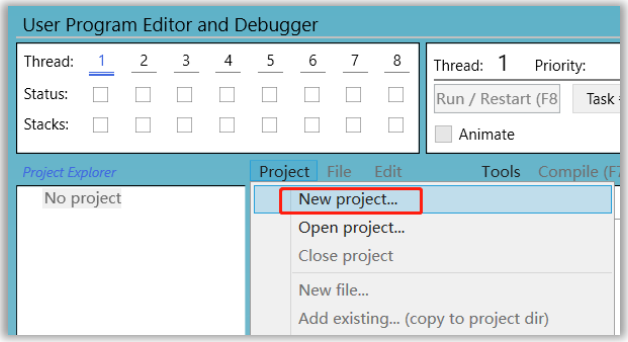
PCSuite features a powerful and comprehensive integrated development environment, IDE+, designed for developing control programs and running control units independently. Our user programs support multi-threaded tasks running simultaneously (AGD series supports up to 8 threads, AGM800 supports up to 12 threads). The intuitive scripting language facilitates the writing of if/for/while/switch statements. Users can also define their own variables and expressions, set interrupt event functions, and even use C-like pointers to read variables. Depending on specific products and applications, our controllers can execute up to 500 low-level instructions within 1us. The IDE+ also supports creating multiple task folders, allowing users to group frequently used functions into separate folders for easy reuse. The IDE+ debugger supports single-step execution, breakpoints, and monitoring windows for easy debugging.

2 Getting to Know PCSuite IDE Programming Environment

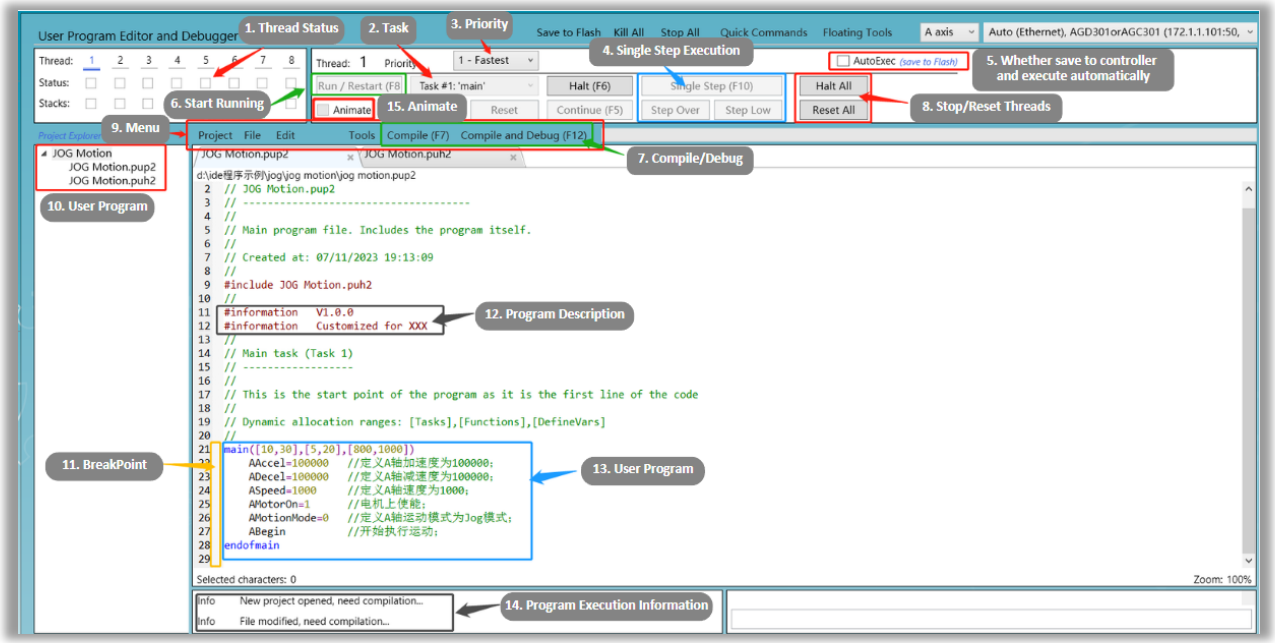
- Switch Menu Bar: **PROGRAM → IDE+**, enter the IDE User-Program programming environment interface



- Click Project: Project→New project to create a new User-Program file, which will automatically create a folder named after the project in the user-selected directory (see section 3.1 for details)



2.1 IDE Programming Environment Interface



1. Thread Status: The controller supports multiple threads running simultaneously. The status of each thread is displayed here. When a thread is not running, it appears white. When a thread is active, it appears green, and it turns red if there is an error. Move the mouse over the status "□" below the corresponding thread to display error information, allowing the user to troubleshoot based on this information.

2. Task: Displays the tasks defined in the User-Program (after compilation). Users can select the corresponding task and click "Run/Restart" to run the task.

3. Priority: Defines the priority of the main thread task. For other thread or task priorities, refer to section 3.9.

4. Single Step Execution: Click "Single Step" to execute User-Program command lines sequentially.

5. Auto Execution: Check this option to automatically run the program upon power-up.

6. Start Running: After successful compilation, click "Run/Restart" to start running the program.

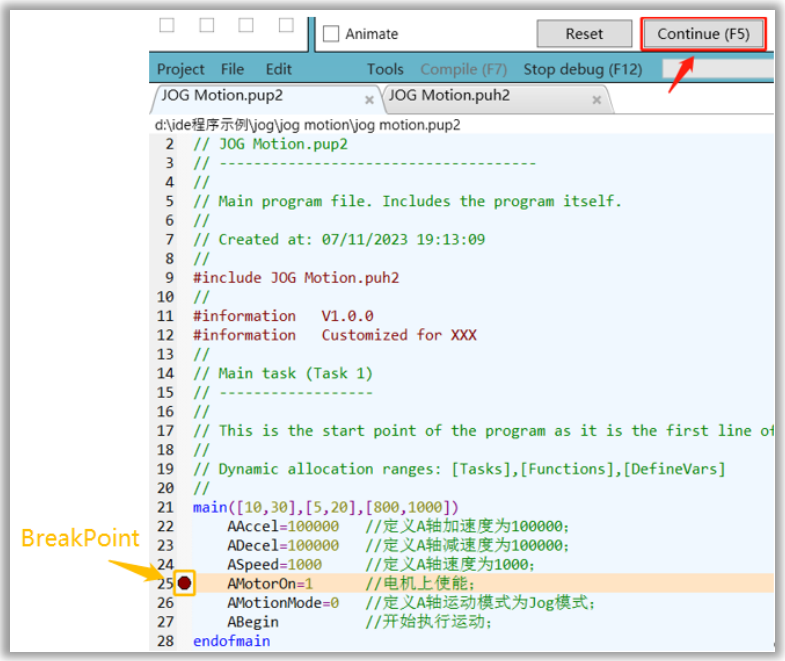
7. Compile/Debug: After writing the user program, compile it to generate the lower-level program. If there are syntax errors, they will be indicated during compilation. Note that when compiling, if the motor is enabled, PCSuite will prompt the user to disable the motor.

8. Stop/Reset Threads: "Halt All" stops all threads, with the User-Program command "AprogHalt-All." "Reset All" resets all threads, with the User-Program command "AProgResetAll."

9. Menu: Includes operations such as creating, opening, and closing project files, as well as tools like compiling, downloading, and clearing breakpoints.

10. User Program: Displays the user project, including *.pup2 and *.puh2 files. The user program is contained in the *.pup2 file, while user-defined variables and pointers are in the *.puh2 file.

11. BreakPoint: After successful compilation, click the breakpoint setting area to set breakpoints, indicated by a dot on the left of the line. The program will pause before executing the line. Click "Continue" or press F5 to continue execution. Multiple breakpoints are supported.



12. Program Description: Users can write custom information such as the User-Program version and functionality, saved as strings in the controller and viewable in the info section.

13. User Program: The main program is edited in this area, including functions and tasks.

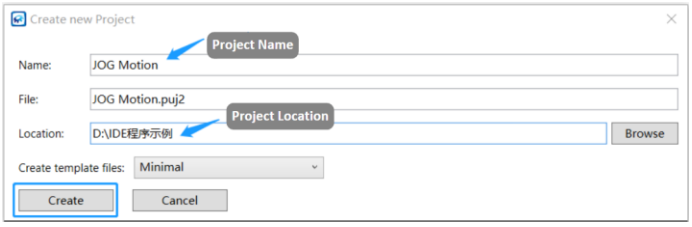
14. Program Execution Information: Displays debugging information during compilation or execution, including error messages.

15. Animate: When checked, the currently running code line is highlighted in real-time during program execution, helping users monitor program flow.

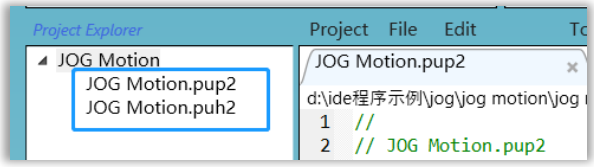
Note: Re-downloading firmware will erase the User-Program stored in flash, requiring re-compilation and saving.

2.2 Program Execution Steps

- Step 1: Click **Project→New Project** to create a new project (see item 9), or click **Project→Open Project** to open a local program.

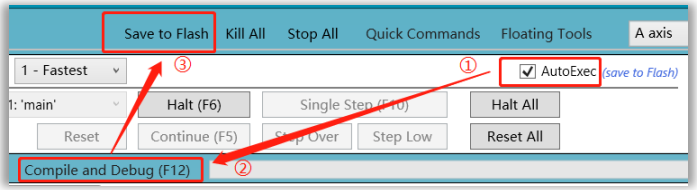


- Step 2: Double-click the *.pup2 file in the Project Explorer to open the main program editing area and write the program. Double-click the *.puh2 file in the variable editing area to define variables (see item 10).



- Step 3: Click "Compile and Debug" (see item 7) to compile and debug the program. If there are no syntax errors, the program will turn light blue.
- Step 4: Click "Run/Restart" (see item 6) to start executing the program.
- Stop and reset thread steps: Stop→Halt All→Reset All

2.3 Saving User-Program to Controller Flash



- Step 1: Click "Project→Open project," double-click the file to open the local IDE User-Program.
- Step 2: Check "AutoExec" (see item 5) in the IDE interface or execute the AAutoExec=1 command in the Terminal.
- Step 3: Click "Compile and Debug" (see item 7) to compile and run the program.
- Step 4: Click "Save to flash" (or press Alt+S) to save to the controller flash. When saved successfully, "Save to flash" will flash on the left side of the PCSuite interface.

After saving the User-Program to the controller, it will automatically execute upon power-up.

Please note: the program saved in the controller is in low-level instructions, while the program in the IDE programming environment is in high-level instructions. **Programs saved in the controller flash cannot be uploaded to PCSuite.** Therefore, it is essential to back up the User-Program source program locally and manage its versions.

3 User Program Basic Syntax

Follow these basic rules when writing User-Programs in the IDE:

- **Keywords**: All keywords must be prefixed with an axis number (even if the keyword is axis-independent). Keywords are case-insensitive.
- **End Marks**: Statements end with a carriage return, line feed, or carriage return + line feed. Each line can only contain one statement (with optional end-of-line comments) and must be within a single line (no continuation lines).
- **Case Sensitivity**: The programming language is case-sensitive.
- **Comments**: Supports single-line comments (//...) and block comments (/.../).
- **Value Range**: All constants or variables and their calculation results only support signed Int32 type data, not floating-point numbers. If the result is not an integer, it will be rounded (different from INT rounding). If the calculation result exceeds the range or violates basic mathematical rules (e.g., division by 0), an error will occur during program execution.

3.1 User Program Files

A new project created in the User Program Editor will automatically create a folder named after the project in the user-selected directory. This folder contains *pup2, *puh2, and *puj2 files.

- ***.pup2**

  User programs are contained in *pup2 files. Currently, User_Program can only include one *pup2 file and multiple *puh2 files. The *pup2 file name must match the project name, and only the project's .pup2 file will be used during compilation.

  *pup2 files can contain theoretically unlimited comments to enhance the readability of the program. Comments will be removed during the compilation process, and the *.cup2 file will not include comments (or any data not needed for user program execution).
- ***.puh2**

  Contains user-defined constants and variables.
- ***.puj2**

  An extensible markup file (similar to XML).

Upon compilation, three files will be automatically generated in the project folder: *.cup2, *.cupb2, and *_DefineVars.h.

- ***.cup2**

  The output file during compilation is saved with the *.cup2 extension. This is the file downloaded to the controller, containing statements converted to be executable by the controller's CPU, along with additional information used by the controller to optimize (size and speed) the execution of the user program.
- ***.cupb2**

  Internally generated to enable the compilation and debugging process.
- ***_DefineVars.h**

  Internally generated to enable the compilation and debugging process.

3.2 Compiler Directives

The user program includes compiler directives to extend its functionality.

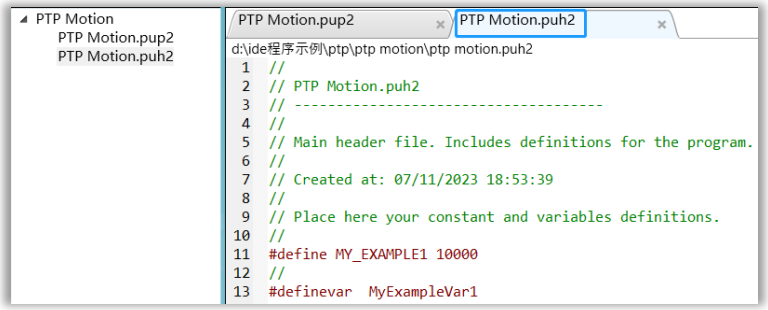
Example:

- Define Constants/Variables: #define

Syntax: `#define <variable_name> < variable_value>`

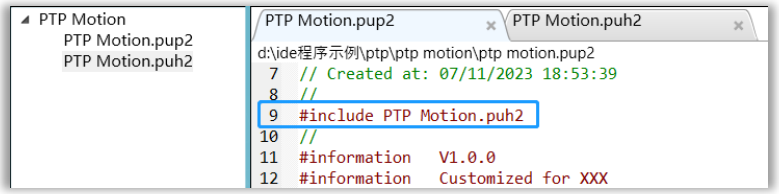
- Define Variable: #definevar

  Syntax: `#definevar <variable_name>`



- Include Header File: #include. A Project can contain multiple *.puh2 files.

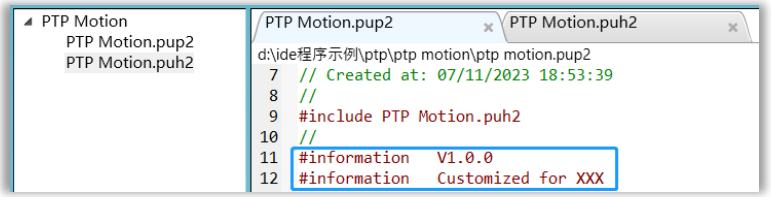
  Syntax: `#include <puh2_name.puh2>`



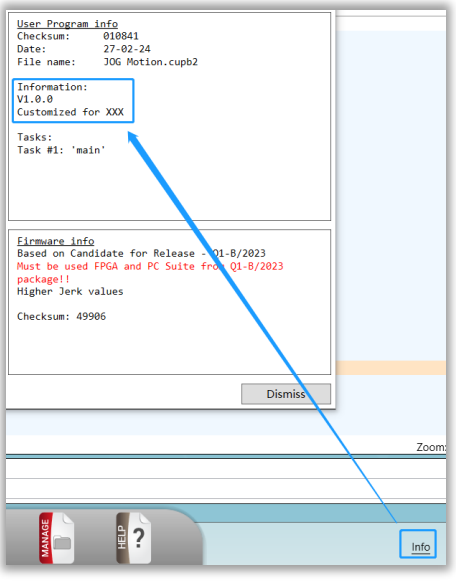
- Load Information: #information

  Download non-volatile memory from the controller, such as User-Program version information, program remarks, etc.

  Syntax: `#information <information_content>`



Read the content of information in the controller.



3.3 Constants and Variables

In the User-Program, the constants or variables used only support integer data (signed 32-bit integer, with values ranging from -2^{31}~2^{31}-1). Therefore, users should be mindful of the possible size of the operation result when performing constant or variable assignments to avoid runtime errors. Variable names can include letters, numbers, and underscores but must start with a letter and be longer than 2 characters.

Define Constant: `#define <variable_name> < variable_value>`

Define Variable: `#definevar <variable_name>`

Example of a linear equation:

```
#define slop_k 10        // Define the slope of the linear equation as 10
#define intercepts_b 50  // Define the intercept of the linear equation as 50

#definevar X1            // Define variable X1
#definevar Y1            // Define variable Y1
```

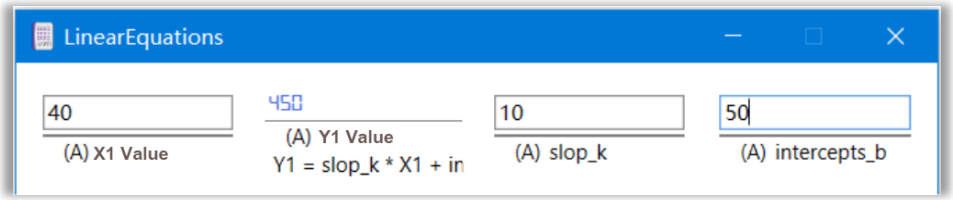
```
while (1)
    X1 = AUserParam[5]    // For demonstration purposes, use UserParam[5] as the input variable
    Y1 = slop_k * X1 + intercepts_b
    AUserParam[6] = Y1    // For demonstration purposes, store the output variable in UserParam[6]
end
```

After running the above program: The result will be calculated based on the parameter values given by the user.

Of course, the user can also associate the value of constants with UserParam or GenData elements. In this case, the corresponding UserParam or GenData element values can be changed in real-time to alter the value of the constants:

```
#define slop_k AUserParam[7]       // Associate slop_k with AUserParam[7]
#define intercepts_b AUserParam[8] // Associate intercepts_b with AUserParam[8]
```

Result of the operation:



(Please refer to the PCSuite UserPanels feature)

Of course, User-Program also supports hexadecimal number assignment and operations, with results displayed in decimal: `0x<HEX_value>.`

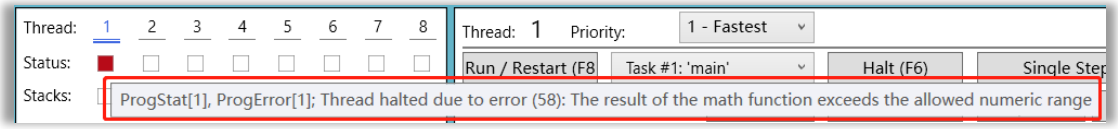
For example: AGenData[502] = 0xA+0xB, the calculation result is AGenData[502]=21

The following program will produce an error:

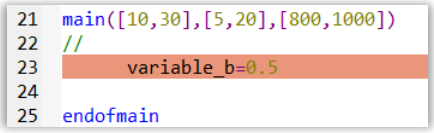
- Data out of range (runtime error)

```
main([10,30],[5,20],[800,1000])
//
    variable_a=power(2,31)    // Assign the value of 2^31 to variable_a

endofmain
```

- Error message



- Unsupported data type (compilation error)



3.4 Array

In the Agito controller's memory, some general-purpose 1-dimensional arrays are allocated, including arrays such as GenData[*] and UserParam[*]. The sizes of these arrays may vary depending on the controller model and firmware version. Users can directly assign values to, and perform operations on, these arrays in the User-Program. It is worth noting that GenData[*] is a general-purpose array, and using certain features (such as ECAM) will occupy the contents of these arrays.

In the User-Program, custom arrays are also allowed, supporting 1D, 2D, and 3D arrays:

1D Array: `#definevar <Array_name>[<Array_size>]`

2D Array: `#definevar <Array_name>[<X_Array_size>][<Y_Array_size>]`

3D Array: `#definevar <Array_name>[<X_Array_size>][<Y_Array_size>][<Z_Array_size>]`

It is worth noting that regardless of whether it is a 1D, 2D, or 3D array, the total length of the array cannot exceed the maximum limit. For example, for the AGD301 controller, it should not exceed 192, which means (X_Array_size* Y_Array_size* Z_Array_size) ≤ 192

Variable names can consist of letters, numbers, and underscores, but they must begin with a letter and have a length of more than 2 characters.

In the following example, a custom array named Array_a of size 10 is defined Array_a[10], and its values are assigned to AGenData[501-510]:

Define Array in *.puh2:

```
#definevar Array_a[10]    // Define array
#definevar iCounter
```

Define Array in *.pup2:

```
iCounter = 1
for (iCounter=1;iCounter<=10; iCounter++)
    Array_a[iCounter]=power(iCounter,2)   // Starting from iCounter=1, assign iCounter^2 to each element of the array
    AGenData[500+iCounter]=Array_a[iCounter]  // Assign the values of Array_a[1-10] to AGenData[501-510]
end
```

Result of the operation:

```
------------------------------------------------------------------------
AGenData[501-510]
501: 1>; 502: 4>; 503: 9>; 504: 16>; 505: 25>; 506: 36>; 507: 49>; 508: 64>; 509: 81>; 510: 100>
------------------------------------------------------------------------
```

Users can also associate variables with specific elements of GenData[*] or UserParam[*]:

```
#define MyVal_1 AGenData[500]  // Associate variable MyVal_1 with GenData[500]

#definevar MyVal_2{502}        // Associate variable MyVal_2 with GenData[502]
```

Define Array in *.puh2:

```
#define <MyVal_name> AGenData[<index>]  or #definevar <MyVal_name>{<index>}
```

Of course, you can also directly assign variables to GenData[*] in the User-Program:

```
15  #define variable_c AGenData[501]
```

```
21  main([10,30],[5,20],[800,1000])
22  //
23      AGenData[501]=200
24      AGenData[502]=0
25      AGenData[502]=variable_c
26
27  endofmain
```

Result of the operation:

```
Agendata[502]
200>
```

3.5 Pointer

User-Programs in the IDE support C-like pointers, allowing direct access to variable addresses.

Define Pointer:

```
#defineptr <Pointer_name> <Pointer_Keyword>
```

Get Pointer Address:

```
& <Pointer_Keyword>
```

Operate on Address Value:

```
* <Pointer_Keyword>
```

Example 1: Obtaining pointer address and assigning values

```
AGenData[500] = &AEventTable[1]  // Get the address of AEventTable[1] and store it in AGenData[500]
AGenData[501] = *AGenData[500]   // Get the value stored at the address pointed to by AGenData[500]
```

Example 2: Define a pointer to EventTable and assign values to it:

Define Pointer in *.puh2:

```
16  #defineptr EventTable_ptr AEventTable[1]
17  #definevar counter
```

Define Pointer in *.pup2:

```
21  main([10,30],[5,20],[800,1000])
22  //
23  // Place here the main code of your program
24  //
25      for(counter=1;counter<=5;counter++)
26          SetEventTable(counter,counter*10)
27      end
28  endofmain
29
30
31  function:SetEventTable(index,val)
32      *(EventTable_ptr+(index-1)*65536)=val    //&AEventTable[1]
33  endoffunc
```

Result of the operation:

Position Events Table (width in [us])

	Value [user-units]	Select	Corrected Value
[1]	10	1	0
[2]	20	1	0
[3]	30	1	0
[4]	40	1	0
[5]	50	1	0
[6]	0	1	0
[7]	0	1	0
[8]	0	1	0

3.6 Conditional Statements

User-Programs support if, switch, and nested if/switch statements.

- **If Statement**

  Syntax:

```
if (boolean_expression)
   /* Statements to execute if boolean_expression is true */
end
```

(For examples, refer to section 4.3 in this document)

- **If...Else Statement**

  Syntax:

```
if (boolean_expression)
   /* Statements to execute if boolean_expression is true */
else
   /* Statements to execute if boolean_expression is false */
end
```

Of course, an if can also be followed by multiple **else if** conditions:

```
if (boolean_expression 1)
   /* Execute if boolean_expression 1 is true */
else if (boolean_expression 2)
   /* Execute if boolean_expression 2 is true */
else if (boolean_expression 3)
   /* Execute if boolean_expression 3 is true */
else
   /* Execute if none of the above conditions are true */
end
```

Once an else if condition is satisfied, the remaining else if or else statements will not be executed

(For examples, refer to section 4.5 in this document)

- **Switch Statement**

  Syntax:

```
switch(expression)
    case constant-expression
       statement(s)
       break
    case constant-expression
       statement(s)
       break
    /* Any number of case statements can be inserted */
    default /* Optional */
       statement(s)
       break
end
```

(For examples, refer to section 4.8 in this document)

3.7 Loop Statements

User-Programs support nested loop statements.

- **While Loop**

Syntax:

```
while (condition)
   statement(s)
end
```

- **For Loop**

  Syntax:

```
for (init; condition; increment)
   statement(s)
end
```

(For examples, refer to section 4.6 in this document)

3.8 Operators

- **Arithmetic Operations**

Operator	Description	Rule
+	Adds two operands	A+B, note that the sum must not exceed the Int32 range
-	Subtracts the second operand from the first	A-B, note that the difference must not exceed the Int32 range
*****	Multiplies two operands	A*B, note that the product must not exceed the Int32 range
/	Divides numerator by denominator	A/B, denominator must not be 0, the quotient is rounded down
()	Parentheses	Perform operations within parentheses first
%	Modulus operator, remainder after division	A%B
++	Increment operator, increases integer value by 1	A++
--	Decrement operator, decreases integer value by 1	A--

- **Basic Functions**

Function	Description	Rule
power	Returns the power of a given number	=power(<value>, <power>), number is the base, power is the exponent, e.g., power(2, 3) = 8
abs	Absolute value	=abs(<value>), abs(-2) = 2
sqrt	Square root	=sqrt(<value>), sqrt(9) = 3

- **Bitwise Operations**

Bitwise operations are commonly used for judging or operating on digital signals or system states.

Operator	Description	Rule
&	Bitwise AND	A & B, refer to the examples in section 4.5 of this document
|	Bitwise OR	A | B, refer to the examples in section 4.5 of this document
^	Bitwise XOR	A ^ B, copies a bit if it is set in one operand but not both
~	Bitwise NOT (including the sign bit)	~ B, results in the complement form of a signed binary number
<<	Left shift	A << B, shifts the value of A to the left by B bits
>>	Right shift	A >> B, shifts the value of A to the right by B bits

- **Relational Operations**

Operator	Description	Rule
==	Equal	True if the values on the left and right are equal
!=	Not equal	True if the values on the left and right are not equal
>	Greater than	True if the value on the left is greater than the value on the right
<	Less than	True if the value on the left is less than the value on the right
>=	Greater than or equal to	True if the value on the left is greater than or equal to the value on the right
<=	Less than or equal to	True if the value on the left is less than or equal to the value on the right

- **Logical Operations**

Operator	Description	Rule
&&	Logical AND	True if both operands are non-zero
||	Logical OR	True if either of the operands is non-zero
!	Logical NOT	Reverses the logical state of its operand; True becomes False and vice versa

- **Assignment Operations**

Operator	Description	Rule
=	Simple assignment	C = A + B assigns the value of A + B to C
+=	Add and assign	C += A is equivalent to C = C + A
-=	Subtract and assign	C -= A is equivalent to C = C - A
*=	Multiply and assign	C *= A is equivalent to C = C * A
/=	Divide and assign	C /= A is equivalent to C = C / A

3.9 Functions

Each User-Program contains a main function, which will be assigned to Task 1 by the CPU and cannot be pre-empted.

```
21  main([10,30],[5,20],[800,1000])
22  //
23  // Place here the main code of your program
24  //
25  endofmain
```

Users can define multiple custom functions in the User-Program and call these functions within threads or tasks.

- **Function Without Parameters**

```
function:<function_name>
   /*function_body*/
endoffunc
```

- **Function with parameters and no return value**

```
function:<function_name>( param_1, param_2, param_3,...)
   /*function_body*/
endoffunc
```

Call: `<function_name>( argum_1, argum_2, argum_3,...)`

- **Function with parameters and return value**

```
function:return_1,return_2,... = <function_name>( param_1, param_2, param_3,...)
   ...
   return_1=statement(s)
   return_2=statement(s)
   ...
endoffunc
```

Call: `return_1,return_2,... = <function_name>( argum_1, argum_2, argum_3,...)`

Example 1:

Ternary Sum Function (with parameters and no return value)

```
21  main([10,30],[5,20],[800,1000])
22  //
23      Sum(30,50,80)
24  //
25  endofmain
26
27  function: Sum(X1,X2,X3)
28      AGenData[500]=X1+X2+X3
29  endoffunc
```

Result: AGenData[500] = 180

Example 2:

Function Returning the Sum and Product of Three Numbers (with parameters and return values)

```
main([10,30],[5,20],[800,1000])
while (1)
    a1=AGenData[501]      // Used only for demonstration
    b1=AGenData[502]      // Used only for demonstration
    c1=AGenData[503]      // Used only for demonstration
    sum1,product1=Sum(a1,b1,c1)  // Call function
    AGenData[504]=sum1          // Used only to demonstrate the result
    AGenData[505]=product1      // Used only to demonstrate the result
end
endofmain

function: sum,product=Sum(X1,X2,X3) // Define function
    sum=X1+X2+X3                    // First return value
    product=X1*X2*X3                // Second return value
    return                          // return is optional
endoffunc
```

3.10 Tasks and Threads

User-Program supports multitasking and multithreading, allowing custom task names, thread allocation, and setting task priorities.

- **Define Task**

```
task:Task_name{<Task#>}
   /*Task_body*/
endoftask
```

- **Bind Thread**

```
AProgRun[<Thread#>],<Task#>
```

Example:

```
main([10,30],[5,20],[800,1000])
    AGenData[497]=0
    AGenData[498]=0
    AGenData[499]=0
    AGenData[500]=0
    AGenData[501]=0
    AProgRun[5], 2       // Bind Task 2 to Thread 5
    AProgRun[6], 3       // Bind Task 3 to Thread 6
    while (1)
        X1=AGenData[497]  // Assign value of AGenData[497] to variable X1
        X2=AGenData[498]  // Assign value of AGenData[498] to variable X2
        X3=AGenData[499]  // Assign value of AGenData[499] to variable X3
    end

//
endofmain

/* Function to calculate the sum of X1, X2, and X3 */
function: Sum(X1, X2, X3)
    AGenData[500]=X1+X2+X3
endoffunc

/* Function to calculate the product of X1, X2, and X3 */
function: Product(X1, X2, X3)
    AGenData[501]=X1*X2*X3
endoffunc

task: SumX1_X2_X3{2}   // Assign Task SumX1_X2_X3 to Task 2
    while (1)
        Sum(X1,X2,X3)
    end
endoftask

task: ProductX1_X2_X3{3}  // Assign Task ProductX1_X2_X3 to Task 3
    while (1)
        Product(X1,X2,X3)
    end
endoftask
```

The above program will calculate the sum and product of the values X1, X2, and X3 in real-time and output the results to AGenData[500] and AGenData[501].

- **Priority**

IDE User-program supports user-defined thread and task priorities, with a total of 10 priorities available, ranging from 1 to 10 (highest to lowest). By default, all thread priorities are set to "1-Fastest". When defining priorities, you can use the following format:
Define thread priority: `AProgPriority[<Thread#>] = <Priority#>`
Define and call task: `runtask(<Name>, <Thread#>, <Priority#>)`

- **Restart and Reset Threads**

When errors or exceptions occur during the User-Program execution, you can restart and reset the threads:

1. **Stop all threads:**

```
23      AProgHaltAll
```

Resetting requires sending a reset command to each thread individually: AProgReset[<Thread#>]

2. **Restart user-specified thread:**

```
44  AProgRun[2], 2      //Run thread_2
45  AProgRun[3], 3      //Run thread_3
46  AProgRun[4], 4      //Run thread_4
47
48  while (1)
49      if  ((AProgError[2]!=0) && (AGenData[808]==2))   //AGenData[808]=2, reset thread_2#
50              AProgHalt[2]
51              AProgReset[2]
52              AWaitTime, 1000
53              AProgRun[2],2
54              AGenData[808]=0
55          end
56      if ((AProgError[3]!=0) && (AGenData[808]==3))   //AGenData[808]=3, reset thread_3#
57              AProgHalt[3]
58              AProgReset[3]
59              AWaitTime, 1000
60              AProgRun[3],3
61              AGenData[808]=0
62          end
63      if ((AProgError[4]!=0) && (AGenData[808]==4))   //AGenData[808]=4, reset thread_4#
64              AProgHalt[4]
65              AProgReset[4]
66              AWaitTime, 1000
67              AProgRun[4],4
68              AGenData[808]=0
69          end
70  end
```

4 User Program Examples

The following examples help users quickly master the PCSuite IDE programming environment and write complex user programs based on project requirements.

4.1 PTP Motion

- **Absolute Target Position Motion**

```
main([10,30],[5,20],[800,1000])
    AAccel=100000       // Set axis A acceleration to 100000;
    ADecel=100000       // Set axis A deceleration to 100000;
    ASpeed=1000         // Set axis A speed to 1000;
    ARelTrgt=0          // Set axis A relative target position to 0;
    AAbsTrgt=2000       // Set axis A absolute target position to 2000;
    AMotorOn=1          // Enable motor A;
    AMotionMode=1       // Set axis A motion mode to PTP mode;
    ABegin              // Start motion;

    while (AInTargetStat!=4)  // Check if the motion has reached the target position;
    end
endofmain
```

- **Relative Target Position Motion**

```
main([10,30],[5,20],[800,1000])
    AAccel=100000       // Set A axis acceleration to 100000
    ADecel=100000       // Set A axis deceleration to 100000
    ASpeed=1000         // Set A axis speed to 1000
    ARelTrgt=3000       // Set A axis relative target position to 3000
    AAbsTrgt=0          // Set A axis absolute target position to 0
    AMotorOn=1          // Enable the motor
    AMotionMode=1       // Set A axis motion mode to PTP mode
    ABegin              // Start motion

    while (AInTargetStat!=4) // Check if the motion has reached the target position
    end

endofmain
```

4.2 Jog Motion

```
main([10,30],[5,20],[800,1000])
    AAccel=100000       // Set A axis acceleration to 100000
    ADecel=100000       // Set A axis deceleration to 100000
    ASpeed=1000         // Set A axis speed to 1000
    AMotorOn=1          // Enable the motor
    AMotionMode=0       // Set A axis motion mode to Jog mode
    ABegin              // Start motion
endofmain
```

4.3 Sending Command Commutation

```
main([10,30],[5,20],[800,1000])
//
    // If the motor starts commutation,
    // delay approximately 5s before commutation to allow the controller to initialize, then proceed
    AWaitTime, 5000

    if (AComtStatus[1]!=100)   // Check if axis A has completed commutation, if not, execute commutation
        AComtMode[5]=1282            // Execute commutation command
        AWaitTime, 2000             // Delay wait
        while (AComtStatus[1]==1)  // Pause during commutation process
        end
        AWaitTime, 500
        if (AComtStatus[1]==-3)    // If commutation fails, retry once
            AComtMode[5]=1282
            while (AComtStatus[1]!=100)
            end
        end
    end
    AWaitTime, 1000

    if (BComtStatus[1]!=100)  // Check if axis B has completed commutation, if not, execute commutation
        BComtMode[5]=1282          // Execute commutation command
        AWaitTime, 2000            // Delay wait
        while (BComtStatus[1]==1)  // Pause during commutation process
        end
        AWaitTime, 500
        if (BComtStatus[1]==-3)    // If commutation fails, retry once
            BComtMode[5]=1282
            while (BComtStatus[1]!=100)
            end
        end
    end
    AWaitTime, 1000

    while ((AComtStatus[1]+BComtStatus[1])!=200)  // Check if both axis A and B have completed commutation
    end
//
endofmain
```

4.4 Homing

```
main([10,30],[5,20],[800,1000])
    // Call the function to set homing parameters for axis A.
    // Of course, you can choose not to use the function and directly place all parameters here.
    AHomingParaDef()

    AHomingOn=1                // Send homing command to axis A
    while (AHomingStat!=100)  // Pause during the homing process
    end

endofmain

function: AHomingParaDef()  // Define function to set homing parameters for axis A
    AHomingDef[1]=8
    AHomingDef[2]=1
    AHomingDef[3]=16384
    AHomingDef[4]=0
    AHomingDef[5]=0
    AHomingDef[6]=0
    AHomingDef[7]=0
    AHomingDef[8]=0
    AHomingDef[9]=0
    AHomingDef[10]=0
    AHomingDef[11]=14
    AHomingDef[12]=1
    AHomingDef[13]=31
    AHomingDef[14]=3277
    AHomingDef[15]=0
    AHomingDef[16]=0
    AHomingDef[17]=0
    AHomingDef[18]=0
    AHomingDef[19]=0
    AHomingDef[20]=0
    AHomingDef[21]=15
    AHomingDef[22]=200000
    AHomingDef[23]=200000
    AHomingDef[24]=2000000
    AHomingDef[25]=983040
    AHomingDef[26]=0
    AHomingDef[27]=0
    AHomingDef[28]=0
    AHomingDef[29]=0
    AHomingDef[30]=0
    AHomingDef[31]=16
    AHomingDef[32]=200000
    AHomingDef[33]=2000000
    AHomingDef[34]=2000000
    AHomingDef[35]=81920
```

```
    AHomingDef[36]=0
    AHomingDef[37]=0
    AHomingDef[38]=0
    AHomingDef[39]=0
    AHomingDef[40]=0
    AHomingDef[41]=6
    AHomingDef[42]=0
    AHomingDef[43]=16384
    AHomingDef[44]=0
    AHomingDef[45]=0
    AHomingDef[46]=0
    AHomingDef[47]=0
    AHomingDef[48]=0
    AHomingDef[49]=0
    AHomingDef[50]=0
    AHomingDef[51]=0
    AHomingDef[52]=0
    AHomingDef[53]=0
    AHomingDef[54]=0
    AHomingDef[55]=0
    AHomingDef[56]=0
    AHomingDef[57]=0
    AHomingDef[58]=0
    AHomingDef[59]=0
    AHomingDef[60]=0
endoffunc
```

4.5 Digital I/O Handling

Digital signals in Agito controllers start from bit_0 incrementally. Digital_Input_1 / Digital_Output_1 corresponds to bit_0, and so on.

- **Check Digital Input Status**

```
if (((ADInPort >> DInPort_bit)&1) == 1)   //Check digital input bit [DInPort_bit] Status
/* When the digital input bit [DInPort_bit] is ON, proceed the code here*/
else if (((ADInPort >> DInPort_bit)&1) == 0)
/* When the digital input bit [DInPort_bit] is OFF, proceed the code here*/
end
```

- **Invert Digital Input Logic**

```
ADInLog = ADInLog ^ (1 << DInLog_bit)    //Invert the digital input bit [DInLog_bit]
```

- **Change Digital Output Status**

```
// Method #1:
ADOutPort = ADOutPort | (1 << DOutPort_bit)    //Set digital output bit [DOutPort_bit] to ON
ADOutPort = ADOutPort & (~(1 << DOutPort_bit)) //Set digital output bit [DOutPort_bit] to OFF

// Method #2:
ADOutPortSBit[DOutPort_num]  //Set digital output number [DOutPort_num] to ON, [DOutPort_num] starts from 1
ADOutPortCBit[DOutPort_num]  //Set digital output number [DOutPort_num] to OFF, [DOutPort_num] starts from 1
```

- **Invert Digital Output Logic**

```
// Method #1:
ADOutPort = ADOutPort ^ (1 << DOutPort_bit)  //Invert the digital output bit [DOutPort_bit]

// Method #2:
ADOutPortTBit[DOutPort_num] //Invert the digital output number [DOutPort_num], [DOutPort_num] starts from 1
```

4.6 Error Compensation Laser Program

*LaserCalibration.pup2 | LaserCalibration.puh2 | Variables

Main Program

```
#definevar Rept_Times
#definevar Motion_Times
```

```
// Laser calibration example prog

main([10,30],[5,20],[800,1000])
    AGenData[500]=0                 // Initialize AGenData[500] to 0
    AHomingOn=1                     // Homing on
    while (AHomingStat!=100)        // Wait for homing to complete
    end
    AMotorOn=1                      // Turn on the motor
    ARelTrgt=0                      // Set relative target to 0
    AAbsTrgt=0                      // Set absolute target to 0
    ASpeed=10000                    // Set laser calibration motion speed
    AAccel=100000                   // Set laser calibration acceleration
    ADecel=100000                   // Set laser calibration deceleration
    ABegin                          // Start motion
    while (AGenData[500]==0)  // Wait for indication, when AGenData[500] is set to 1, the program will proceed
    end
    for (Rept_Times=1; Rept_Times<=3; Rept_Times++)  // Total number of motions
        for (Motion_Times=1; Motion_Times<=12; Motion_Times++)  // Number of forward motions
            ARelTrgt=10000          // Set relative target distance
            ABegin                  // Start motion
            AWaitStatus[7],0        // Wait for motion result
            AWaitTime, 3000         // Wait 3 seconds after motion result
        end
        AWaitTime, 3000             // Wait 3 seconds after all forward motions
        for (Motion_Times=1; Motion_Times<=12; Motion_Times++)  // Number of backward motions
            ARelTrgt=-10000         // Set relative target distance
            ABegin                  // Start motion
            AWaitStatus[7],0        // Wait for motion result
            AWaitTime, 3000         // Wait 3 seconds after motion result
        end
        AWaitTime, 3000             // Wait 3 seconds after all backward motions
    end
    AGenData[500]=0                 // Reset AGenData[500] to 0
endofmain
```

4.7 Multi-threading Calls

```
main([10,30],[5,20],[800,1000])
    AProgRun[5], 2            // Bind task 2 to thread 5
    AWaitTime, 500
    AProgRun[6], 3            // Bind task 3 to thread 6
    AWaitTime, 500
    AProgRun[7], 4            // Bind task 4 to thread 7
    AWaitTime, 500
endofmain

task: A_AxisMotion{2}         // Bind the first task to task 2, note task 1 is reserved for the main thread, custom use is not allowed
    AMotorOn=1
endoftask

task: B_AxisMotion{3}         // Bind the second task to task 3
    BMotorOn=1
endoftask

task: AGenData_Def{4}         // Bind the third task to task 4
    AGenData[1000]=100
endoftask
```

4.8 Gear Mode Gear Ratio Switching

```
14  #definevar GearTemp
```

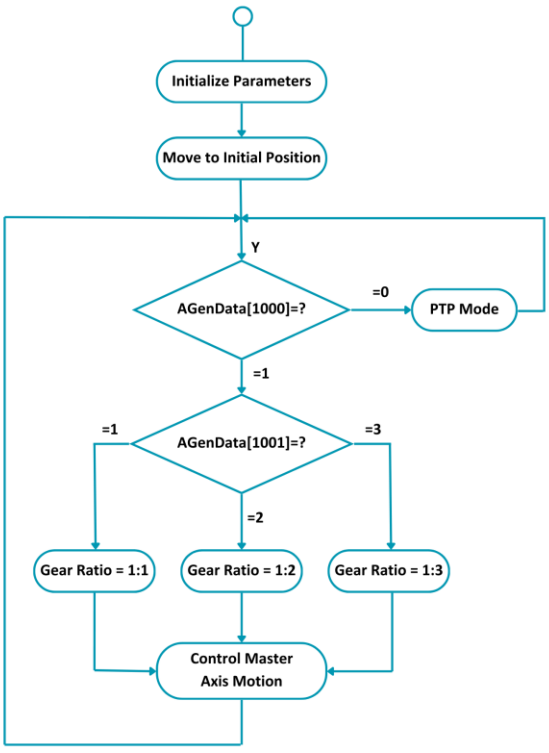
// Define a variable in .puh2 to store the current gear ratio coefficient

```
////////////////////////////////////
// Initialize variables
AGenData[1000]=0
AGenData[1001]=1
AGenData[1002]=0
GearTemp=0

// Move to initial position
AMotionMode=1
ARelTrgt=0
AAbsTrgt=1000
ASpeed=5000
AAccel=50000
ADecel=50000
BRelTrgt=0
BAbsTrgt=2000
BSpeed=5000
BAccel=50000
BDecel=50000
AMotorOn=1
BMotorOn=1
ABegin
BBegin
```

```
// Check if A and B axes have reached the target position
while ((AInTargetStat!=4) || (BInTargetStat!=4))
end

while ((AConFlt==0) && (BConFlt==0))
    if (AGenData[1000]==0)
        if (BMotionMode==5)
            BStop
            while (BMotionStat!=0)
            end
            GearTemp=0
            BMotionMode=1
        end
    else if (AGenData[1000]==1)
        if (GearTemp!=AGenData[1001])
            switch (AGenData[1001])
```

```
                case 1
                    BStop
                    while (BMotionStat!=0)
                    end
                    BMotionMode=5
                    BMasterFact=65536
                    GearTemp=AGenData[1001]
                    BBegin
                    break
                case 2
                    BStop
                    while (BMotionStat!=0)
                    end
                    BMotionMode=5
                    BMasterFact=131072
                    GearTemp=AGenData[1001]
                    BBegin
                    break
                case 3
                    BStop
                    while (BMotionStat!=0)
                    end
                    BMotionMode=5
                    BMasterFact=196608
                    GearTemp=AGenData[1001]
                    BBegin
                    break
                end
            end
        end
end
////////////////////////////////////////////////////
```

Program Logic Flowchart:



4.9 Event Functions

In certain situations, it is necessary to execute internal or external time on a specific user program function (e.g., digital signal changes, internal state changes, sensor values, etc.) within the controller.

Users can link the User-Program with predefined event functions. When an event is triggered, the user program execution will jump (similar to an interrupt) to this event function and return to the position before the event was triggered once the event function ends.

Event functions can only run in the User-Program main thread (i.e., thread 1). Therefore, users need to ensure that the main thread is always running when using Event functions (e.g., the infinite loop while(1) in lines 43-44 in the example). Once the User-Program jumps to the Event function, all other threads will continue running as before, but users can include commands in the event function to stop other threads or modify their priorities.

The User-Program supports up to 5 Event functions (this may vary depending on the product model and firmware version). Event #1 is a fast event, while #2~#5 are regular events. Users can fully customize the characteristics of each Event, including which controller parameter to use (ProgEventPar), mask (ProgEventMask), trigger type (ProgEventType), and trigger value (ProgEventVal).

The following example demonstrates the implementation of continuous PEG activation in ModRev mode using an Event function. It re-enables the PEG function every time it crosses the re-count position (zero position), achieving continuous PEG signal output in ModRev mode.

```
21  main([10,30],[5,20],[800,1000])
22
23      AProgEventGEn = 0
24      AProgEventOn = 0            /*to be sure it is disabled while configuring.This is the master on/off of the
25                                  evenets detection. If it is 0, teh controller do nothing for events*/
26
27      AProgEventMask[1] = -1      /*AND mask before checking the trigger. -1 means 0xFFFFFFFF, so like no mask.
28                                  Mask is used for looking at specific bits, like given input bit. */
29
30      AProgEventType[1] = 5       /*Type of trigger, just like in data recording,5 is rising edge*/
31
32      AProgEventVal[1] = 100      //The value of the trigger. Must be rising edge througth value of 100
33
34      AProgEventPar[1] = &APos    /*This is a 'pointer' to any variable (parameter/keyword) of the controller.
35                                     Just any of them. Here it points to APos (position of A axis)*/
36      AProgEventEn[1] = 1         // Enable event #1 (specific enable).
37
38      AProgEventGEn = 1           /*This is teh main events of the interrupts (maine nabling of looking for
39                                   events flags at the user program engine). Here: enabled.*/
40      AProgEventOn = 1            /*Enable the feature. The controller will start looking fr events (in the interrupt it will
41                                   look fr event occournces, and also when running user program, it will look for event flags)*/
42
43      while (1)                   //  A loop of doing almost nothing, and waiting for events
44      end
45
46  endofmain
```

```
47
48
49  function: MyEventFunc1() [Event#1]
50      if (AEventOn==0)
51          AEventOn=1
52      end
53
54  endoffunc
```

4.10 Sine Wave Motion

Sine Wave.pup2 主程序 | Sine Wave.puh2

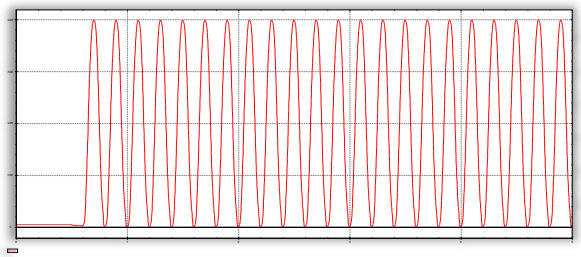
d:\ide程序示例\sine wave\sine wave\sine wave.pup2

```
main([10,30],[5,20],[800,1000])
//
    ARptWait = 0                            // Don't wait between point to point motion
    AJerk = 0
    AmpIn = 200
    FreqIn = 5                              // No "smoothing" of profile needed
//
    while(1)
        if(SineEnable==1)                    //If UpdateVal=1, enable sine wave
            AStop
            AWaitTime, 500                  // Allow time to stop, wait 500[ms]

            AAccel = 32*AmpIn*FreqIn*FreqIn     // Calculated needed acceleration for quasi-sinusoidal motion
            ADecel = AAccel                     // Calculated needed deceleration for quasi-sinusoidal motion
            ASpeed = 8*AmpIn*FreqIn             // Calculated needed speed for quasi-sinusoidal motion
            AMotionMode = 1                     // Non-repetitive point to point motion
            AAbsTrgt = 0                        // Go to predefined 0
            AMotorOn = 1
            ABegin
            while (AInTargetStat != 4)          // Wait until motor has reached its target
                AWaitTime, 100
            end
            AMotionMode = 2                     // Repetitive point to point motion
            AAbsTrgt = 2*AmpIn                  // Full range is twice the required amplitude
            ABegin
            SineEnable = 0
        else if (SineEnable==2)                  //If UpdateVal=2, disable sine wave
            AStop
            SineEnable = 0
        end

    end
//
endofmain
```

Sine Wave.puh2 变量

d:\ide程序示例\sine wave\sine wave\sine wave.puh2

```
//
// Sine Wave.puh2
// -------------------------------------
//
// Main header file. Includes definitions for the program.
//
// Created at: 07/29/2020 09:54:50
//
// Place here your constant and variables definitions.
//
#define MY_EXAMPLE1 10000
//

#definevar  SineEnable{797}
#definevar  FreqIn{798}
#definevar  AmpIn{799}
```



4.11 Typical Force Control

- **Closed-loop Force Control**

```
CGoToPosMode // Switch to position mode to prevent collisions caused by enabling force control mode while in a suspended state.
while (COperationMode == 4) // Ensure switching to position mode
    CGoToPosMode
    CWaitTime, 500
end
```

1. Ensure switching to position mode before enabling

```
    CMotionMode = 1 // Switch to PTP control mode
    CRelTrgt = 0
    CAbsTrgt = 3800 // Set an absolute target position that will definitely encounter the object under pressure
    CSpeed = 20000 // Set the speed for high-speed approach
    CAccel = 200000 // Set acceleration for high-speed approach
    CDecel = 200000 // Set deceleration for high-speed approach
    CJerk = 0

    CMotorOn = 1 // Enable the motor

    CSpeedChgNew = 500 // Set the speed after switching to low speed
    CSpeedChgPos = 3400 // Set the position after switching to low speed
    CSpeedChgDir = 0
    CSpeedChgOn = 1 // Turn on the switch for the low-speed switching function
    CBegin          // Begin PTP Motion
```

2. Set relevant parameters for high-speed and low-speed approach in position mode

```
    CCurrPosTh = 3500 // Set the condition value for switching to force control
    CCurrPosThDir = 1
    CForceAInTh = 5 // Set the force feedback threshold for switching to force control

    CForceCmdSrc = 1 // Set force control command mode to "Scheduled Force Command Values" mode

    CForceGain = 50000 // Set PID parameters for closed-loop force control, the same below
    CForceKi = 5
    CForceKd = 0
    CForceI = 0
    CForceVelFFW = 0
```

3. Set conditions for switching to close-loop force control and the PID parameters for force closed-loop

```
CForceCmdVal[1] = 150 // Set the first segment of the force control command to 150 force-units
CForceCmdHTime[1] = 500 // Set the duration of the first segment of the force command to 500 ms
CForceCmdSlope[1] = 1000 // Set the slope of the first segment of the force command to 1000 force-units
CForceCmdVal[2] = 170 // Set the second segment of the force control command to 170 force-units
CForceCmdHTime[2] = 500 // Set the duration of the second segment of the force command to 500 ms
CForceCmdSlope[2] = 1000 // Set the slope of the second segment of the force command to 1000 force-units
CForceCmdVal[3] = 120 // Set the third segment of the force control command to 120 force-units
CForceCmdHTime[3] = 500 // Set the duration of the third segment of the force command to 500 ms
CForceCmdSlope[3] = 1000 // Set the slope of the third segment of the force command to 1000 force-units
CForceCmdHTime[4] = 0 // Set the duration of the fourth segment of the force command to 0, execute only 3 segments of force control command
```

4. Set the command parameters for closed-loop force control

```
CBeginOnToPos = 1 // Enable retract function after switching back from force control mode to position mode.
CRetractTarget = 0 // Set the retraction target position
CRetractSpeed = 20000 // Set the retraction speed
```

5. Set retract movement after the end of force control command

- **Open-loop Force Control**

```
CGoToPosMode // Switch to position mode to prevent collisions caused by enabling force control mode while in a suspended state.
while (COperationMode == 1) // Ensure switching to position mode
    CGoToPosMode
    CWaitTime, 500
end
```

1. Ensure switching to position mode before enabling

```
CMotionMode = 1 // Switch to PTP control mode
CRelTrgt = 0
CAbsTrgt = 3800 // Set an absolute target position that will definitely encounter the object under pressure
CSpeed = 20000 // Set the speed for high-speed approach
CAccel = 2000000 // Set acceleration for high-speed approach
CDecel = 2000000 // Set deceleration for high-speed approach
CJerk = 0

CMotorOn = 1 // Enable the motor

CSpeedChgNew = 500 // Set the speed after switching to low speed
CSpeedChgPos = 3400 // Set the position after switching to low speed
CSpeedChgDir = 0
CSpeedChgOn = 1 // Turn on the switch for the low-speed switching function
CBegin          // Begin PTP Motion
```

2. Set the motion parameters for fast and slow approach to the object under pressure

```
CCurrPosTh = 3500 // Set the condition value for switching to force control
CCurrPosThDir = 1
CCurrCurrTh = 120 // Set the current command threshold for switching to force control
CCurrCurrThDir = 0

CCurrCmdSrc = 1 // Set force control command mode to "Scheduled Current Command Values" mode

CCurrGain = 770 // Set PID parameters for open-loop force control, the same below
CCurrKi = 358
```

3. Set the conditions for switching to open-loop force control and the PID parameters

```
CCurrCmdVal[1] = 150 // Set the first segment of the force control command to 150 force-units
CCurrCmdHTime[1] = 500 // Set the duration of the first segment of the force command to 500 ms
CCurrCmdSlope[1] = 1000 // Set the slope of the first segment of the force command to 1000 force-units
CCurrCmdVal[2] = 170 // Set the second segment of the force control command to 170 force-units
CCurrCmdHTime[2] = 500 // Set the duration of the second segment of the force command to 500 ms
CCurrCmdSlope[2] = 1000 // Set the slope of the second segment of the force command to 1000 force-units
CCurrCmdVal[3] = 120 // Set the third segment of the force control command to 120 force-units
CCurrCmdHTime[3] = 500 // Set the duration of the third segment of the force command to 500 ms
CCurrCmdSlope[3] = 1000 // Set the slope of the third segment of the force command to 1000 force-units
CCurrCmdHTime[4] = 0 // Set the duration of the fourth segment of the force command to 0, execute only 3 segments of force control command
```

4. Set the command parameters for open-loop force control

```
CBeginOnToPos = 1 // Enable retract function after switching back from force control mode to position mode.
CRetractTarget = 0 // Set the retraction target position
CRetractSpeed = 20000 // Set the retraction speed
```

5. Set the retreat command after switching back to position mode from open-loop force control

5 Introduction to Common Keywords

The following introduces only some commonly used keywords. For detailed usage of other keywords, please refer to the "Akribis-Agito Controller Keywords Reference" document.

In the syntax format, Axis_index represents the axis number, such as A for axis A, B for axis B, and so on.

5.1 Commutation Related

- **ComtMode**:
  ComtMode [] is an array used to control the commutation process.

Keyword	ComtMode
Type	Parameter
Syntax	Read: Axis_index+ComtMode[Array_index] Write: Axis_index+ComtMode[Array_index]=Value
Value Type	Int
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	No
Save To Flash	Yes
Array Index Range	1:24
Axis Related	Yes
Common Command	ComtMode[5]=1282, restart commutation operation

- **ComtStatus**: Returns the commutation status.
  ComtStatus[] is used to return the commutation status.

Keyword	ComtStatus
Type	Parameter
Syntax	Read: Axis_index+ComtStatus[Array_index]
Value Type	Int
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Array Index Range	1:2
Axis Related	Yes
Value meaning	Axis_index ComtStatus[1]=100, indicates successful commutation.

5.2 Homing Related

The homing-related parameters are mainly defined by the following keywords:

- **HomingOn**

At power-on or after a reset, the value of HomingOn will be set to 0. When the user sets it to 1, the controller will execute the homing operation according to the defined homing sequence. After the homing operation is completed (whether successful or with an error), the value of HomingOn will be reset to 0.

Keyword	HomingOn
Type	Parameter
Syntax	Read: Axis_index+HomingOn Write: Axis_index+HomingOn=Value (Value=0,1)
Value Type	Int: 0~1
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes
Value meaning	Axis_index HomingOn=1, start Homing operation.

- **HomingDef**

HomingDef is used to define the parameters for the homing process, including steps and types of parameters. This parameter is generally exported through the PCSuite homing tool (for detailed information, please refer to the "Agito Homing User Manual").

Keyword	HomingDef
Type	Parameter
Syntax	Read: Axis_index+HomingDef[Array_index] Write: Axis_index+HomingDef[Array_index]=Value
Array Index Range	[1:150]
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes
Value meaning	HomingDef[1]: Defines the step type. HomingDef[2-10]: Define the parameters for step 1. HomingDef[21-30]: Define the parameters for step 2, and so on.

- **HomingStat**

HomingStat is used to describe the homing status.

Keyword	HomingStat
Type	Parameter
Syntax	Read: Axis_index+HomingStat
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No

Keyword	HomingStat
Axis Related	Yes
Value meaning	▪ 0: No homing was done after power on or reset. ▪ Positive values (except 100) : Homing in progress, HomingStat value indicates the current step of the homing process. ▪ -1 : Homing failed due to HomingDef parameter error. Each step's related parameters are checked before the homing process starts. ▪ -2 : Homing failed due to timeout. Each step has a defined execution delay time, and an error is reported if the step is not completed within that time. ▪ -3 : Homing failed due to unexpected motor off during the homing process. The axis was disabled due to some fault (reflected at the value of ConFlt) ▪ -4 : Homing failed due to motion error, such as undefined motion end method (RLS, index, reached target, etc). ▪ -5 : Homing failed due to an unrecognized step type, such as HomingDef[1]=50. ▪ -6 : Homing failed because the motor was detected to be in motion during step transition. ▪ -7 : Homing failed due to an incorrect step, meaning the last step of the homing program is not "0-End Homing". For example, the last step being HomingDef[61]=1 is incorrect. ▪ -8 : Homing failed due to unexpected limit. ▪ 100 : Homing completed successfully.

5.3 Motion Related

- **Abort**

  Set the speed to 0 to stop the motion immediately (the motor remains enabled).

Keyword	Abort
Type	Command
Syntax	Axis_index+Abort
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **AbsTrgt**

  AbsTrgt is used to set the absolute target position for PTP (point-to-point) and reciprocating motion. This target position is independent of the current position, and AbsTrgt defines where the motion will stop. For PTP motion, this position is where the motor will stop. For reciprocating PTP motion, the AbsTrgt position will be one target point of the PTP motion, with the current position being the other target point for reciprocating motion**. If RelTrgt (relative target position) is not 0, AbsTrgt is ignored, and RelTrgt is used to execute the next target action**. AbsTrgt can be modified during motion and will be executed immediately.

Keyword	AbsTrgt
Type	Variable
Syntax	Read: Axis_index+AbsTrgt
Array Index Range	-2,147,483,648~2,147,483,647
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

Example:

- If the current position Pos = 1000, AbsTrgt = 5000, and RelTrgt = 0, the next PTP motion will stop at Pos = 5000.
- If the current position Pos = 1000, AbsTrgt = 5000, and RelTrgt = 7000, the next PTP motion will stop at Pos = 8000 (target position = Pos (current) + RelTrgt).
- If the current position Pos = 1000, AbsTrgt = 5000, and RelTrgt = 0, the next reciprocating PTP motion will first move to Pos = 5000 and then back to Pos = 1000, repeating the above motion.

See also : **RelTrgt**

- **Accel**

Accel is used in all motions to represent acceleration in user units/sec^2. The value of Accel can be changed during motion and will be executed immediately. For example, if the motion is currently accelerating, the acceleration will be changed by the current command. If the target speed has been reached, the new acceleration will be applied in the next acceleration motion.

Keyword	Accel
Type	Variable
Syntax	Read: Axis_index+Accel
Array Index Range	100~2,000,000,000
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

See also : Vel, Speed, Decel, MaxAcc

- **Begin**

Begin command is used to start motion after all motion parameters (such as motion mode, acceleration/deceleration, speed, etc.) have been set. This command must be issued after the motor is enabled; otherwise, an error will occur. The type of motion to begin is defined by MotionMode. After sending the "Begin" command, motion parameters (speed, acceleration/deceleration, etc.) can be changed in real-time and will take effect immediately.

Keyword	Begin
Type	Command
Syntax	Axis_index+Begin
Access	Read only

Keyword	Begin
Allowed to Write During Motion	No
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **BeginOnToPos**

  BeginOnToPos command is used to change the control mode to position mode during motion, mainly applied in force control.

Keyword	BeginOnToPos
Type	Parameter
Syntax	Read: Axis_index+BeginOnToPos Write: Axis_index+BeginOnToPos=Value, Value=0,1
Array Index Range	0, 1
Default Value	0
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Decel**

  Decel is used to define the deceleration for all motions, measured in user units/sec^2. Decel can be changed during motion, and the updated value will take effect immediately. If the motion is currently decelerating, the deceleration rate will change.

  Stop command stops the motion at the deceleration defined by **Decel**, while an emergency stop uses the deceleration defined by **EmrgDec**.

Keyword	Decel
Type	Parameter
Syntax	Read: Axis_index+Decel Write: Axis_index+Decel=Value
Array Index Range	100~2,000,000,000
Default Value	100,000
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **AuxPos**

  AuxPos provides feedback on the auxiliary encoder reading.

Keyword	AuxPos
Type	Parameter

Keyword	AuxPos
Syntax	Read: Axis_index+AuxPos Write: Axis_index+AuxPos=Value
Array Index Range	-2,147,483,648~2,147,483,647
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	No
Save To Flash	No
Axis Related	Yes

- **CurrRef**

CurrRef returns the loop current of the controller (command current), measured in mA.

Keyword	CurrRef
Type	Parameter
Syntax	Read: Axis_index+CurrRef
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **IndexPos**

IndexPos holds the position of the last detected index signal (main encoder).

Keyword	IndexPos
Type	Parameter
Syntax	Read: Axis_index+IndexPos
Array Index Range	-2,147,483,648~2,147,483,647
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **InTargetStat**

InTargetStat indicates the progress of the motion until the target is reached. When the position error in user units is within the maximum allowable position error (InTargetTol) for the target arrival time (InTargetTime), InTargetStat will return the target position reached.

Keyword	InTargetStat
Type	Parameter
Syntax	Read: Axis_index+InTargetStat
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes

Keyword	InTargetStat
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes
Value meaning	▪ 0: Disable ▪ 1: Enable ▪ 2: In motion ▪ 3: Waiting for InTargetTime to end ▪ 4: Target position reached

- **InTargetTime**

  InTargetTime indicates the time (in ms) that the motor should reach the target position within the allowed error range. If the position error is within the InTargetTol for the duration of InTargetTime, InTargetStat returns the target reached.

Keyword	InTargetTime
Type	Parameter
Syntax	Read: Axis_index+InTargetTime
Array Index Range	0~1,000
Default Value	3
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Jerk**

  Jerk defines the time required to establish the desired acceleration/deceleration based on trajectory planning.

  The time to establish acceleration/deceleration is calculated as:

  Jerk Time = SAMPLE_TIME * 2^{Jerk}

  where SAMPLE_TIME is the sampling time, and Agito controller sampling time is 1/16384 = 61us.

  In the PCSuite interface, JerkTime is displayed.

  Note that Jerk cannot be changed during motion. A higher Jerk value results in a longer motion time, but the motion will be smoother, and overshoot is usually reduced. Adjust the Jerk value according to the specific application to achieve optimal performance.

Keyword	Jerk
Type	Parameter
Syntax	Read: Axis_index+Jerk
Array Index Range	0~9
Default Value	0
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	Yes
Save To Flash	No

Keyword	Jerk
Axis Related	Yes

- **ModRev**

  ModRev defines the modulus for controller feedback. When this value is 0, the feature is disabled. When set to M, the encoder value will only vary within the range [0, M] counts. If it exceeds M, it will reset and recount from 0 counts.

  This allows for infinite motion in the same direction ([0, M] within the range [RevPLim, FwdPLim]).

  Note: Do not use ModRev and Shaping functions simultaneously.

  Using SetPosition to manually set a position value outside the ModRev range may result in unexpected behavior.

  Infinite motion (ModRev not 0) can be used with smoothing (Jerk not 0) simultaneously, but ModRev and Jerk values must be set correctly. The time to move the ModRev distance at maximum speed should be greater than JerkTime.

  For example: if the maximum speed is 10,000,000 [counts/sec], Jerk = 9 (i.e. 2^9 = 512 samples), and the controller's sampling rate is 16,384 samples/sec, then ModRev must be greater than 10,000,000/16,384*512 = 312,500 [counts].

  The product of Jerk($2^{Jerk[samples]}$) and ModRev ([counts]) sample counts must be less than (2^{31}-1).

  For example, if Jerk = 9, then ModRev must be less than $\frac{2^{31}-1}{2^9}$≈4,194,304 [counts].

Keyword	ModRev
Type	Parameter
Syntax	Read: Axis_index+ModRev Write: Axis_index+ModRev=Value
Array Index Range	0~2,000,000,000
Default Value	0
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	No
Save To Flash	Yes
Axis Related	Yes

- **MotionMode**

  Used to define the type of motion that will be executed after the next "Begin" command. Note that MotionMode cannot be changed before the current motion ends.

Keyword	MotionMode
Type	Parameter
Syntax	Read: Axis_index+MotionMode Write: Axis_index+MotionMode=Value
Range Index Value	0~18
Default Value	0
Access	Read/Write
Allowed to Write During Motion	No

Keyword	MotionMode
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes
Value meaning	▪ 0: Jog ▪ 1: PTP ▪ 2: Point to Point repetitive ▪ 3: Pulse Direction direct mode ▪ 4: Pulse Direction indirect mode ▪ 5: Gear direct motion ▪ 6: Gear indirect motion ▪ 7: ECAM direct motion ▪ 8: ECAM indirect motion ▪ 9: FIFO motion ▪ 10: Slave position reference ▪ 11: CNC group A motion ▪ 12: Joystick direct motion, input analog in signal representing position reference ▪ 13: Joystick indirect motion, input analog in signal representing position reference, and this profiler can be setting by user ▪ 14: Joystick direct motion, input analog in signal representing velocity reference ▪ 15: Joystick indirect motion, input analog in signal representing velocity reference ,and this profiler can be setting by user ▪ 16: Vector motion ▪ 17: CNC Group B motion ▪ 18: Spline buffer motion

- **MotionReason**

  Return value indicating the reason for the last motion end.

  "Begin" command resets the MotionReason value to 0.

Keyword	MotionReason
Type	Parameter
Syntax	Read: Axis_index+MotionReason
Range Index Value	0~11
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

Keyword	MotionReason
Value meaning	▪ 0 : Current motion not yet finished or motion is normal ▪ 1 : Stopped by Stop command ▪ 2 : Stopped by Abort command ▪ 3 : Stopped by StopRep command ▪ 4 : Stopped due to detecting negative limit switch ▪ 5 : Stopped due to detecting positive limit switch ▪ 6 : Stopped by negative soft limit ▪ 7 : Stopped by positive soft limit ▪ 8 : Stopped due to motor disable ▪ 9 : Stopped by StopECAM command ▪ 10 : Stopped by StopFIFO command ▪ 11 : Stopped due to detecting index (Jogging mode only)

- **MotionStat**

  Returns the current motion status. Each bit in MotionStat represents a different motion status. In some cases, multiple motion statuses may be active, and MotionStat = 0 (i.e., all bits are 0) indicates that the motion has stopped. The value of MotionStat is in decimal, which can be converted to a binary number or used with bitwise AND (&) to indicate the current motion status based on each bit.

12	11	10	9	8	7	6	5	4	3	2	1	0	Bit

Keyword	MotionStat
Type	Parameter
Syntax	Read: Axis_index+MotionStat
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes
Value meaning	▪ Bit 0 : In motion ▪ Bit 1 : Waiting for reciprocating motion ▪ Bit 2 : Reciprocating motion stopped (refer to StopRep command) ▪ Bit 3 : Stop command received ▪ Bit 4 : Accelerating ▪ Bit 5 : Decelerating ▪ Bit 6 : Waiting for smoothing to end ▪ Bit 7 : Stopping ECAM motion ▪ Bit 8 : Stopping FIFO ▪ Bit 9 : Waiting for input (motion paused until rising edge of user-defined input) ▪ Bit 10 : Associated with CNC group A ▪ Bit 11 : CNC group A in motion ▪ Bit 12 : CNC motion stopping

- **MotorOn**

  Command used to control the motor enable/disable status. When the motor is disabled, power is not supplied to the motor, and motion control cannot be performed. Due to faults (see ConFlt), the driver may internally disable the motor. When re-enabling, ConFlt will be cleared. If the fault causing the disable is not cleared, the motor will remain in the disabled state. Some commands can only be used when the motor is disabled.

Keyword	MotorOn
Type	Parameter
Syntax	Read: Axis_index+MotorOn Write: Axis_index+MotorOn=Value
Default Value	0
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes
Value meaning	▪ 0 : Disable ▪ 1 : Enable

- **Pos**

  Returns encoder readings in UsrUnits. If UsrUnits = 65536, Pos is in Counts.

Keyword	Pos
Type	Parameter
Syntax	Read: Axis_index+Pos
Value Index Range	-2,147,483,648~2,147,483,647
Default Value	0
Access	Read only
Allowed to Write During Motion	No
Allowed to Write When Enabled	No
Save To Flash	No
Axis Related	Yes

- **PosRef**

  Represents the user position command in UsrUnits.

Keyword	PosRef
Type	Parameter
Syntax	Read: Axis_index+PosRef
Value Index Range	-2,147,483,648~2,147,483,647
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes

Keyword	PosRef
Save To Flash	No
Axis Related	Yes

- **PosErr**

  Returns the position error value in UsrUnits, which is the difference between the position command and the actual position. If the PosErr value exceeds the value defined by MaxPosErr, the motor will be disabled.

Keyword	PosErr
Type	Parameter
Syntax	Read: Axis_index+PosErr
Value Index Range	-2,147,483,648~2,147,483,647
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **RelTrgt**

  Defines the distance between the current position and the target position for PTP motion and reciprocating PTP motion. If RelTrgt is not 0, AbsTrgt is ignored, and RelTrgt is used to determine the next actual motion target. RelTrgt can be changed during motion and will take effect immediately.

Keyword	RelTrgt
Type	Parameter
Syntax	Read: Axis_index+RelTrgt Write: Axis_index+RelTrgt=Value
Value Index Range	-2,147,483,648~2,147,483,647
Default Value	0
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

  See also : AbsTrgt

- **Speed**

  Used to define the target speed in UsrUnits/Sec. The speed can be changed in real-time during motion. If the current motion is in the acceleration and constant speed phase, the speed will be changed to the updated value.

Keyword	Speed
Type	Parameter

Keyword	Speed
Syntax	Read: Axis_index+Speed Write: Axis_index+Speed=Value
Value Index Range	-60,000,000~60,000,000
Default Value	0
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

- **Vel**

  Array returning encoder speed feedback in three different formats, in UsrUnits/Sec.

  - Vel[1]: Filtered speed value, filter parameters defined by VelFilt.
  - Vel[2]: Raw speed value before filtering.
  - Vel[3]: Average value over 16 sampling periods, approximately 1ms.

Keyword	Vel
Type	Parameter
Syntax	Read: Axis_index+Vel[Array_index]
Value Index Range	-2,147,483,648~2,147,483,647
Default Value	0
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **SpeedChgNew**

  Defines the speed variable that will be assigned to Speed when the trigger condition is met.

Keyword	SpeedChgNew
Type	Parameter
Syntax	Read: Axis_index+SpeedChgNew Write: Axis_index+SpeedChgNew=Value
Value Index Range	-60,000,000~60,000,000
Default Value	0
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

- **Stop**

Command to stop the current motion according to the set deceleration. If not in motion, it has no effect.

Keyword	Stop
Type	Command
Syntax	Axis_index+Stop
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

See also : Abort

- **StopCNCA**

Command to stop the current A group CNC motion.

Keyword	StopCNCA
Type	Command
Syntax	Axis_index+StopCNCA
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **StopRep**

Command to stop reciprocating motion. If this command is sent during other types of motion, it will be ignored. When sent during reciprocating motion, the motion will not decelerate and stop immediately; instead, it will stop at the starting point of the next motion after the current motion ends. To stop the motion immediately, use the "Stop" command.

Keyword	StopRep
Type	Command
Syntax	Axis_index+StopRep
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

5.4 Motor Parameters Related

- **EncRes**

Defines encoder resolution.

Keyword	EncRes
Type	Parameter

Keyword	EncRes
Syntax	Axis_index+EncRes
Access	Read/Write
Allowed to Write During Motion	No
Allowed to Write When Enabled	No
Save To Flash	Yes
Axis Related	Yes

- **Ia**

Returns phase A current value in mA.

Keyword	Ia
Type	Parameter
Syntax	Axis_index+Ia
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Ib**

Returns phase B current value in mA.

Keyword	Ib
Type	Parameter
Syntax	Axis_index+Ib
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **MotorCurr**

The total motor current (sum of all phases) in mA.

Keyword	MotorCurr
Type	Parameter
Syntax	Axis_index+MotorCurr
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Va**

Returns the applied voltage to phase A in millivolts.

Keyword	Va
Type	Parameter
Syntax	Axis_index+Va
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Vb**

Returns the applied voltage to phase B in millivolts.

Keyword	Vb
Type	Parameter
Syntax	Axis_index+Vb
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **Vc**

Returns the applied voltage to phase C in millivolts.

Keyword	Vc
Type	Parameter
Syntax	Axis_index+Vc
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

5.5 I/O Related

- **AInPort**

Returns the analog value for each analog input channel. The unit is related to AInMode, and the number of analog input channels (Port_index) varies across different controllers.

Keyword	AInPort
Type	Parameter
Syntax	Axis_index+ AInPort[Port_index]
Access	Read only
Allowed to Write During Motion	Yes

Keyword	AInPort
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **AOutPort**

Returns the analog value for each analog output channel. The unit is related to AOutMode, and the number of analog output channels (Port_index) varies across different controllers.

Keyword	AOutPort
Type	Parameter
Syntax	Read: Axis_index+ AOutPort[Port_index] Write: Axis_index+ AOutPort[Port_index]=Value
Value Index Range	-12,000~12,000
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

- **DInPort**

Returns a decimal value representing the digital input port status. The number of digital input channels varies across different controllers.

- When the input port is "On", DInPort=$\sum_0^i 2^i$, where i is the bit index. If the input is "Off", it does not participate in the calculation.
- The DInPort return value can be converted to binary to check the status of each individual input.
- Bit_0 (LBS) corresponds to Digital_Input_1, and so on.
- Note that DInLog controls the input logic. For example:
  If DInLog = 4 (only inverts the logic of Digital_Input_3), and the input status is:
  - Digital_Input_1 = On
  - Digital_Input_2 = Off
  - Digital_Input_3 = Off
  - Other inputs are Off
- DInPort will return "1" for bit_0, "0" for bit_1, and "1" for bit_2 (because Digital_Input_3 is actually Off, but the logic is inverted, so it returns "1"), resulting in DInPort=$\sum_0^2 2^i = 2^0+2^2=5$

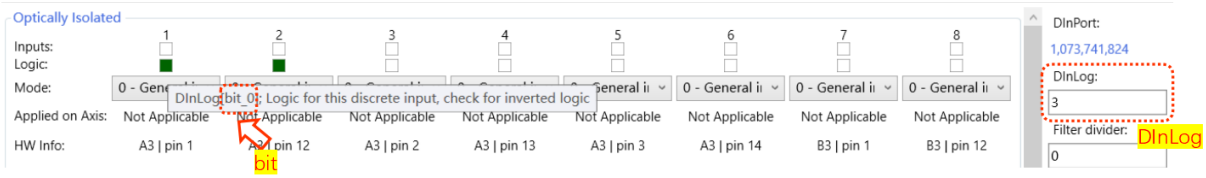
Keyword	DInPort
Type	Parameter
Syntax	Read: Axis_index+DInPort
Value Index Range	-12,000~12,000
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No

Keyword	DInPort
Axis Related	Yes

- **DInLog**

Defines the logic for each digital input channel. If DInLog = 0, when the input port is 1, DInPort returns 1; if DInLog = 1, when the input port is 1, DInPort returns 0.

- When the input port logic is "On", DInLog=$\sum_0^i 2^i$, where i is the bit index. If the logic is "Off", it does not participate in the calculation.
- For example, if only Digital_Input_1 and Digital_Input_2 are enabled, DInLog=$2^0+2^1=3$.



Keyword	DInLog
Type	Parameter
Syntax	Read: Axis_index+DInLog
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

- **DInPortHigh**

Returns a decimal value representing the high-speed digital input port status. The number of digital input channels varies across different controllers.

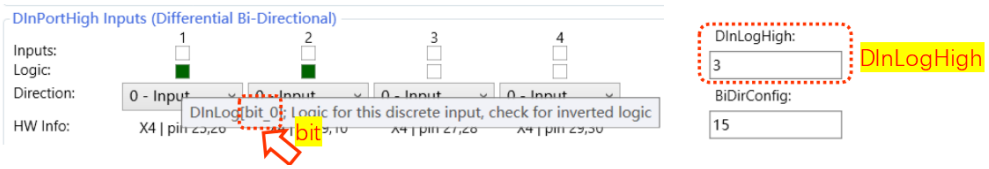
- When the input port is "On", DInPortHigh=$\sum_0^i 2^i$, where i is the bit index. If the input is "Off", it does not participate in the calculation.
- The DInPortHigh value can be converted to binary to check the status of each individual input.
- Bit_0 (LBS) corresponds to DInPortHigh Input_1, and so on.
- Note that DInLogHigh controls the input logic.

Keyword	DInPortHigh
Type	Parameter
Syntax	Read: Axis_index+DInPortHigh[DInPortHigh_index]
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **DInLogHigh**

Defines the logic for each digital input channel. If DInLogHigh = 0, when the input port is 1, DInPort returns 1; if DInLogHigh = 1, when the input port is 1, DInPort returns 0.

- When the input port logic is "On", DInLogHigh=$\sum_0^i 2^i$, where i is the bit index. If the logic is "Off", it does not participate in the calculation.
- For example, if only DInPortHigh Input_1 and DInPortHigh Input_2 are enabled, DInLogHigh = $2^0+2^1=3$.



Keyword	DInLogHigh
Type	Parameter
Syntax	Read: Axis_index+DInLogHigh[DInLogHigh_index]
Access	Read only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	Yes

- **DOutPort**

Returns a decimal value representing the digital output port status. The number of digital output channels varies across different controllers.

- When the output port is "On", DOutPort=$\sum_0^i 2^i$, where i is the bit index. If the output is "Off", it does not participate in the calculation.
- The DOutPort value can be converted to binary to check the status of each individual output.
- The actual output status also depends on other related parameters:
  - The bitwise XOR (XOR) operation is performed between DOutPort's bit_i and DOutLog's bit_i, where i = 0,1,2,....
  - If DOutMode[Port_index] ≠ 0, the output status related to that output will not be affected by the value of the corresponding bit in DOutMode, and the output will only reflect the special function.

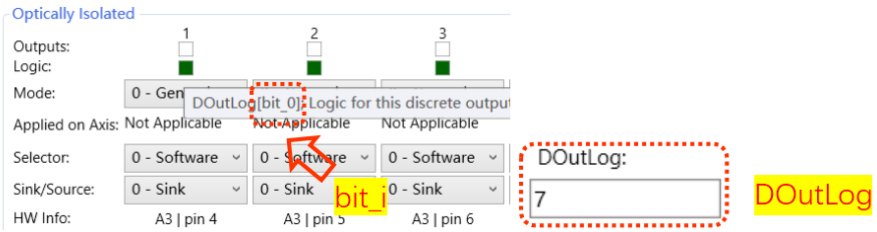
Keyword	DOutPort
Type	Parameter
Syntax	Read: Axis_index+ DOutPort Write: Axis_index+ DOutPort=Value
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

- **DOutLog**

Defines the logic for digital output channels. If DOutLog = 0, when the input port is 1, DOutPort returns 1; if DOutLog = 1, when the input port is 1, DOutPort returns 0.

- When DOutLog ≠ 0 (i.e., enabled), DOutLog=$\sum_0^i 2^i$, where i is the bit index.

- For example, if only the logic of Digital_Output_1, Digital_Output_2, and Digital_Output_3 is set to 1 and the others are set to 0, DOutLog = $2^0+2^1+2^2=7$.



Keyword	DOutLog
Type	Parameter
Syntax	Read: Axis_index+ DOutLog Write: Axis_index+ DOutLog=Value
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	Yes

5.6 System Related

- **WaitStatus**

  WaitStatus is a low-level keyword in UserProgram. The syntax related to "Compare" is usually automatically generated by PCSuite during compilation. It waits until the defined status is met before the user program continues execution.

  Note: This keyword is only allowed for use in the IDE program.

Keyword	WaitStatus	
Type	Command	
Syntax	Axis_index+ WaitStatus[Status type], status value	
Access	Read/Write	
Allowed to Write During Motion	Yes	
Allowed to Write When Enabled	Yes	
Save To Flash	No	
Axis Related	Yes	
Value Meaning	Status type	Meaning
	1	COUNTERDOWN1: Wait until CounterDown[1] reaches the value
	2	COUNTERDOWN2: Wait until CounterDown[2] reaches the value
	3	COUNTERDOWN3: Wait until CounterDown[3] reaches the value
	4	COUNTERDOWN4: Wait until CounterDown[4] reaches the value
	5	COUNTERUP1: Wait until CounterUp[1] reaches the value
	6	COUNTERUP2: Wait until CounterUp[2] reaches the value

Keyword	WaitStatus	
	7	IN_MOTION: Wait until MotionStat in motion bit reaches the value
	8	IN_RPT_WAIT: Wait until MotionStat in repeat wait bit reaches the value
	9	IN_RPT_STOP: Wait until MotionStat stop bit reaches the value
	10	IN_STOP_REQUEST: Wait until MotionStat stop request bit reaches the value
	11	IN_ACCELERATION: Wait until MotionStat in acceleration bit reaches the value
	12	IN_DECELERATION: Wait until MotionStat in deceleration bit reaches the value
	13	IN_WAIT_END_SMOOTH: Wait until MotionStat smoothing ended bit reaches the value
	14	IN_ECAM_STOP: Wait until MotionStat in ECAM stop bit reaches the value
	15	IN_FIFO_STOP: Wait until MotionStat in FIFO stop bit reaches the value
	16	COMMUTATION: Wait until StatReg commutation bit reaches the value
	17	IN_TARGET: Wait until StatReg in target bit reaches the value
	18	REC_TRIG_DETECTED_STATUS: Wait until RecStat trigger detected reaches the value
	19	REC_COMPLETE_STATUS: Wait until RecStat recording complete reaches the value
	20	DIN1: Wait until digital input 1 reaches the value
	21	DIN2: Wait until digital input 2 reaches the value
	22	DIN3: Wait until digital input 3 reaches the value
	23	DIN4: Wait until digital input 4 reaches the value
	24	DIN5: Wait until digital input 5 reaches the value
	25	DIN6: Wait until digital input 6 reaches the value
	26	DIN7: Wait until digital input 7 reaches the value
	27	DIN8: Wait until digital input 8 reaches the value
	28	DIN9: Wait until digital input 9 reaches the value
	29	DIN10: Wait until digital input 10 reaches the value
	30	DIN11: Wait until digital input 11 reaches the value
	31	DIN12: Wait until digital input 12 reaches the value
	32	DIN13: Wait until digital input 13 reaches the value
	33	DIN14: Wait until digital input 14 reaches the value
	34	MOTOR_ON_INPUT_STATUS: Wait until the input designated as motor on reaches the value
	35	VEL_GAIN_CHANGE_STATUS: Wait until the input designated as gain change reaches the value
	36	CLEAR_ABS_ENC_STATUS: Wait until the input designated as clear absolute encoder on reaches the value
	37	CLEAR_IN_PULSES_STATUS: Wait until the input designated clear input pulses on reaches the value
	38	REV_LIMIT_STATUS: Wait until the input designated as reverse limit reaches the value

Keyword	WaitStatus	
	39	FWD_LIMIT_STATUS: Wait until the input designated as forward limit reaches the value
	40	TORQUE_LIMIT_ON_STATUS: Wait until the input designated as torque limit reaches the value
	41	ALARM_RESET_STATUS: Wait until the input designated as alarm reset reaches the value
	42	ABORT_INPUT_STATUS: Wait until the input designated as abort reaches the value
	43	MODE_SWITCH_VEL_POS_STATUS: Wait until the input designated as vel / pos switch reaches the value
	44	MODE_SWITCH_VEL_CUR_STATUS: Wait until the input designated as vel/cur switch reaches the value
	45	MODE_SWITCH_POS_CUR_STATUS: Wait until the input designated as pos/cur switch reaches the value
	46	ADD_FILTER_STATUS: Wait until the input designated as add filter reaches the value

- **GenData**

GenData is a general-purpose array allocated to the user. In this array, users can store integer data, commonly used for storing tables such as ECAM.

Keyword	GenData
Type	Parameter
Syntax	Read: Axis_index+ GenData[Array_index] Write: Axis_index+ GenData[Array_index]=Value
Access	Read/Write
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	Yes
Axis Related	No

- **WaitTime**

WaitTime is used to define the waiting time before continuing to execute the next line of code, measured in milliseconds (ms).

Note: This keyword is only allowed for use in the IDE programming environment.

Keyword	WaitTime
Type	Parameter
Syntax	Write: Axis_index+ WaitTime, Value
Access	Write only
Allowed to Write During Motion	Yes
Allowed to Write When Enabled	Yes
Save To Flash	No
Axis Related	No