Akribis-Agito User Program

User's Manual

Table of contents

General

This file provides the description of Agito-Akribis User Program environment and language.

What is a User-Program?

The User-Program (or script) is the feature that enables the user to download a user-program to a controller and that enables the controller to independently execute the user program upon a proper request by the operator or upon a predefined event within the controller.

Generally speaking, the User-Program provides the ability to stand-alone operation of the controller, without receiving command messages over the communication

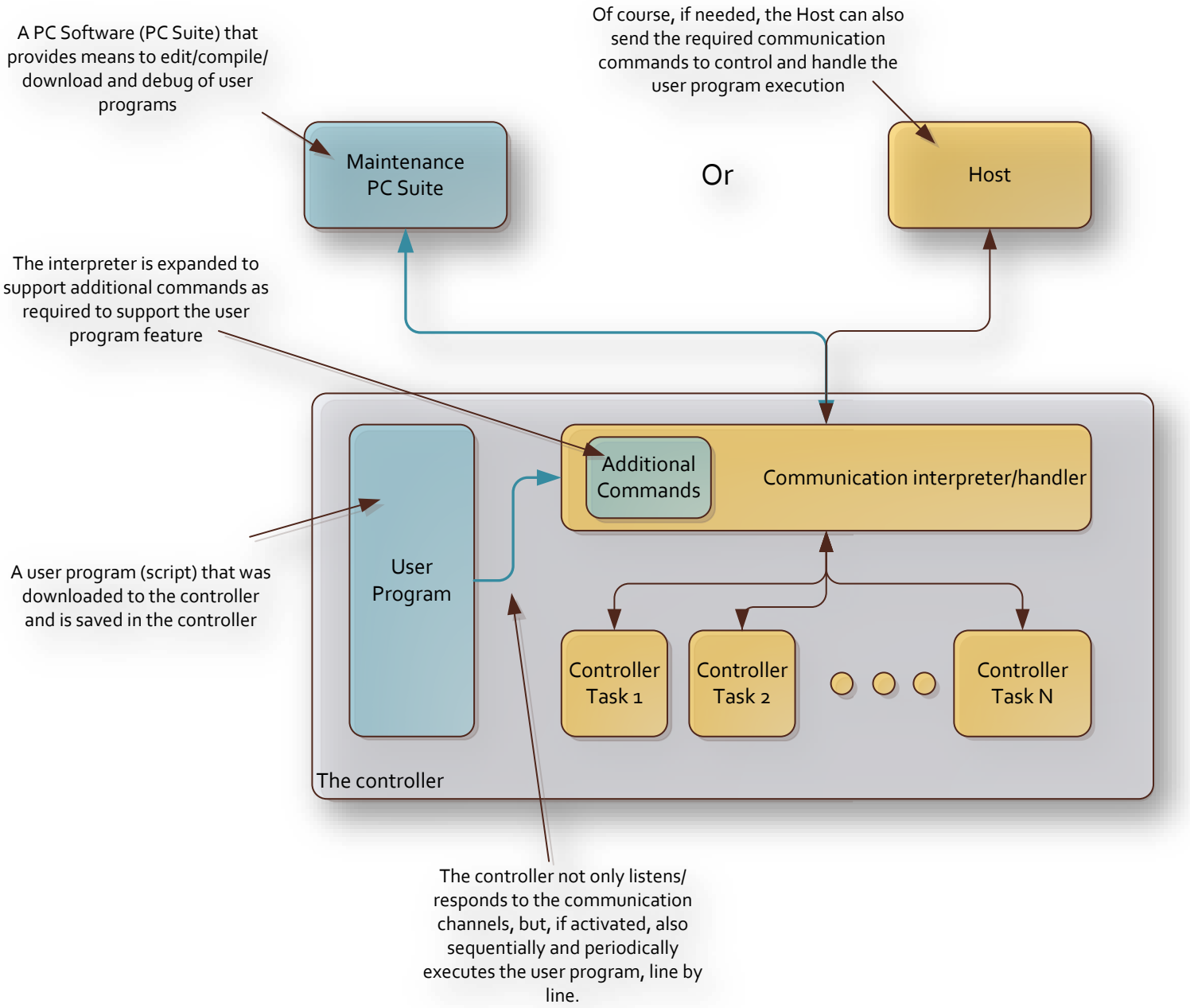
Typical usage of User-Programs

Typical (and partial) usages of a user programs are:

- To execute the machine application.
- To perform homing.
- To perform safety and I/O sequences.
- To execute test scenarios by the hardware engineers.
- To execute test scenarios by the programmers or the field technicians, in the field.
- To collect data about events and processes.
- To support tests of the controller during production (Jigs).

System structure

The figure in the next page presents a system with a controller that supports User-Programs.



Akribis PC Suite software

Akribis PC Suite supports the following additional features in order to establish an environment for the user-programs development:

- Project management:

The PC Suite can handle a user-program project (a collection of files that establish a single program).

- Edit:

  The PC Suite integrates a context sensitive editor for user programs.

- Compile:

  Provide the feature to compile the user program (or better to say: project) into a low level controller user program that can be downloaded into the controller.

  The process can detect errors in the project's files, as well as to properly display the errors and point to their location.

  The compiler also generates all needed information files for the debug process.

> Note:
>
> Debug process is a future feature and is not currently supported by the PC Suite.

- Download:

  The PC Suite supports the process of downloading the controller user program to the controller (in which it is saved into the Flash memory).

- Debug:

  The PC Suite provides the means to debug the user program that is in the controller (execute, halt, single step, breakpoints, and watch variables …). Both the high level and the low level user (see below) programs are shown in debug mode (like C and ASM when debugging embedded C code).

> Note:
>
> Debug process is a future feature and is not currently supported by the PC Suite.

Initially, the "User-Program" is developed (edited, compiled, downloaded, debugged …) using the PC Suite software. Later on, it is downloaded and saved in the controller non-volatile memory and now it can be executed upon a suitable message over the communication channels or it can be executed upon pre-defined built in events in the controller.

The PC Suite is designed to minimize the user program development time; starting writing a new user program till the user program is successfully executed by the controller.

The User-Program language(s)

Generally speaking, the User-Program is a script of commands identical (from the functional point of view) to the commands available over the communication lines. It provides the ability to perform a batch (script) of commands internally by the controller. However, in order to create a meaningful program, the User-Program also supports additional commands, to enable: flow control (if, while, for …), math calculations, user variables, comments, compiler directives etc.

In order to support these additional commands, the controller interpreter is expanded to support the execution of these additional commands from the user program.

The User-Program language is divided into two languages, conceptually similarly to C and ASM. The user develops (writes) the User-Program in the Programmer User-Program Language (*.PUP), which is a high level language, while the controller executes a User-Program in the Controller User-Program Language (*.CUP), which is a lower level language.

The PC software Suite is responsible (during compilation and downloading) to compile the User-Program file that is written in Programmer User-Program Language into a User-Program file that is written on Controller User-Program Language, which is downloaded into the controller.

The usage of the low level Controller User-Program Language enables simple (and small sized) implementation in the controller firmware and ensures fast execution (no need for complex interpretation of high level commands and flow structures during execution).

A User-Program in Programmer User-Program Language can just as well, and transparently, include parts in Controller User-Program Language (just as you can embed ASM code within a C code).

Multiple User-Programs

It is important to note that while a single User-Program is downloaded into the controller, this User-Program can consist of many Tasks (sub-programs) and the user may select to execute any one of them, so that actually the User-Program program may include many programs, each performing a predefined task/process.

Multi-threading

The controller supports the execution of few (controller dependent) User-Program threads in parallel. All threads are executed within a single User-Program (as defined above, it can include multiple User-Program sub-programs/tasks).

Once the controller has executed a command from a given activated thread, it will execute the next command from the next activated thread and so on, in a loop over all the activated threads.

Priority between threads is also supported, so that, for example, a given thread can be executed at the highest rate (command each loop over all the active threads) while another thread will be activated in a lower rate, for example: each 10 loops (actually running 10 times slower compared to the faster thread).

Event functions

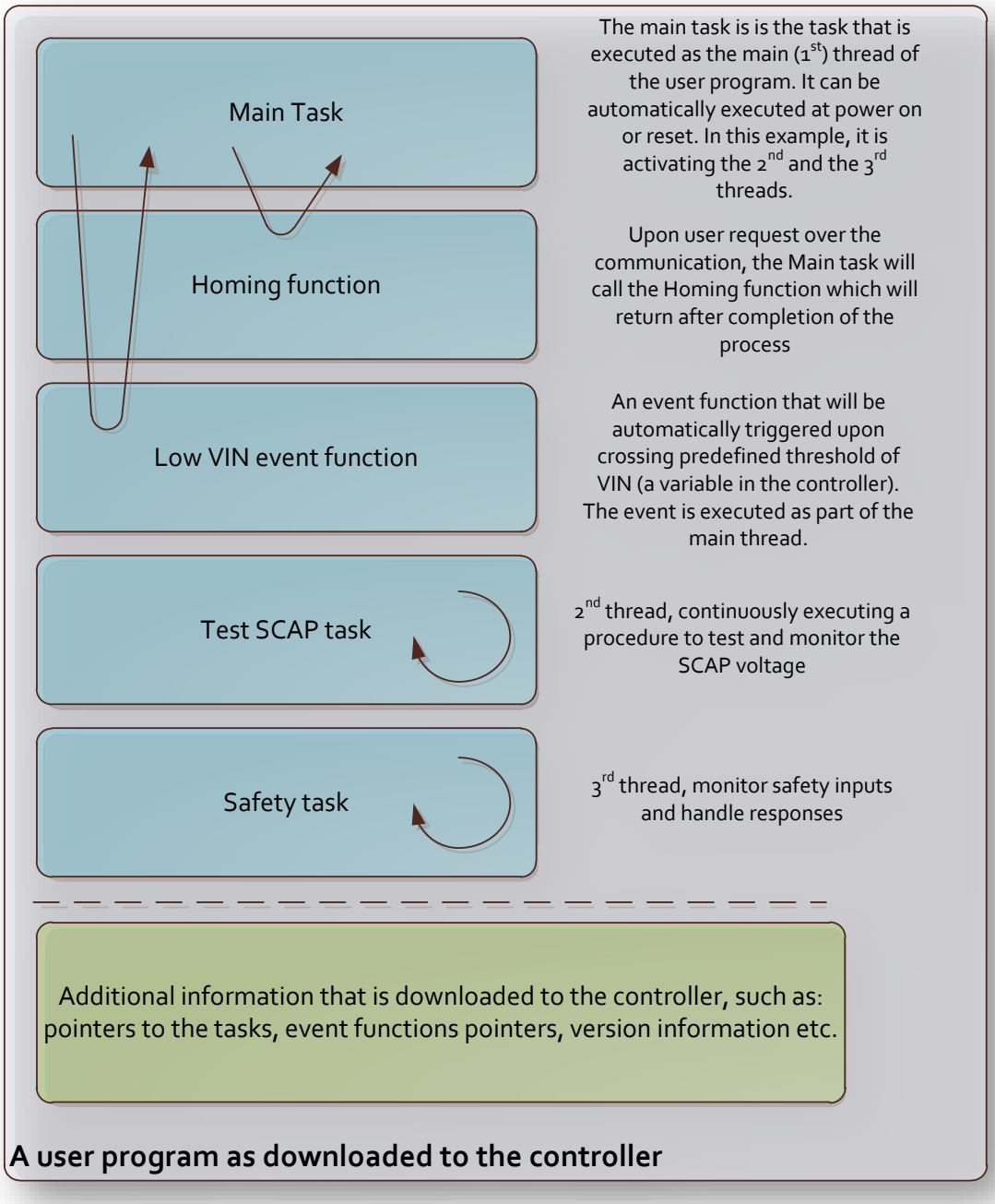
In some cases, it is required that a specific User-Program function will be executed upon a controller internal or external event (change of a discrete input, change of internal state, value of a sensor etc.).

A user may link between a given User-Program function and a user predefined event. In such case, upon a trigger of this event, the User-Program execution will jump (similarly to an interrupt) to this function, from which it will return – at the end of the function - to the same location it was before the event.

Event functions are supported only for the main thread (thread 1) of the User-Program.

Once the User-Program is jumping into an event function, all other threads continue to run as before. However, the user can include, in the event function, the required commands to halt the other threads, or to modify their priority.

The following figure describes the structure of a **sample** user program file and threads execution, including an event function. Note that it is a single project. However, it consists of few Tasks, and the user/programmer can define, for each thread, at which location (Task) to execute.



A user program as downloaded to the controller

Theoretically, as each thread had its own program location pointer, multiple threads can run the same code (task). However, this is not practical in most cases.

Note that the file that is downloaded to the controller (the *.CUP – Controller User Program) includes not only the compiled code of the high level program (the *.PUP file – Programmer User Program), but all the additional information that may be needed by the controller in order to:

- Minimize the size of the related code at the controller.
- Maximize the execution speed of the user program at the controller.

This additional information is prepared by the compiler in the PC Suite and is packed as part of the *.CUP file.

Supported events

The software supports 5 (other numbers can be supported by various products and/or FW versions) event definitions, and the user can fully define the characteristics of each of these events, covering which controller parameter to use, a bitwise mask, trigger type (rising edge, falling edge, equal, not equal…) and a value to look for.

This provides unique flexibility in defining the trigger for a given event function (similar to the definition of data recording trigger).

Note that the triggering of functions upon events is performed only if the main user program thread (Thread 1) is running. It is Thread 1 which will service the Event trigger by calling the Event Function that is tied (linked) to it.

Compiler directives

A User-Program can include compiler directives that are handled by the User-Program compiler to expand the functionality of the User-Program.

Examples are:

#define	(similar to C, described below)
#include	(similar to C, described below)
#information	(described below)
#definevar	(described below)

User-Program files

Note:

Currently, a user program project can consist only a single *.PUP file and multiple *.PUH files (see description below). The *.PUP file must have the same name as the project name.

Multiple *.PUP files within a single project is a future feature.

A User-Program (actually, a project) is constructed from some program files. Each file consists of 1 (or more, although 1 is recommended) Tasks (code segments, each responsible for a given process). A program file may include headers files for compiler directives (only), as described below.

The following file extensions are used:

1. *.pup

   Program files have a *.pup extension. PUP stands for Programmer User-Program language.

   A project may include 1 to many PUP files (each containing at least 1 Task).

2. *.puh

   The *.puh extension is a header files that can be included in a *.pup file.

   Only *.puh files can be included within a *.pup file.

   A PUH file's content is limited to comments and compiler's directives.

   A project may include 1 to many PUH files (each containing at least 1 Task). Only the files that are included by the project's PUP files are actually used in the compilation process.

3. *.cup

   The output file of the compilation process has a *.cup extension. This is the file that is downloaded to the controller. It includes the user program project converted into the Controller User Program language (that can be executed by the controller), as well as additional information that is used by the controller to optimally (size and speed) execute the user program.

Other files and extensions are generated internally by the compiler to enable the compilation and the debug processes.

User variables

Using a built-in array of general parameters (AGenData[] in Agito-Akribis's controllers), the User-Program supports user variables naming, including arrays.

The user can define a name for a variable (together with its location within AGenData[]) and from this point it can refer to the user variable by its name (within the User-Program).

At the moment, only integer (signed 32 bits) variables are supported. Support for floating point variables (and math operations) will be added in the future.

Flow control

The User-Program language is equipped with a wide set of language keywords to support complex flow control of the program.

This includes the keywords: while, if, for … as well as their related keywords such as break, continue, etc.

See details later on within this document.

Version control and information

The User-Program language and environment support the embedding of version related information in both the PUP and the CUP files.

This covers the name of the file, the date/time of its creation, its CRC and any additional information that can be added by the programmer using the #information directive.

This information is available to the user by using a specific communication message that uploads all the version related information from the controller (from the CPU file that was downloaded to the controller).

Comments

The PUP file can include unlimited number of comments, to enhance the readability of the program. Comments are stripped during the compilation process and the CUP file does not include comments (or any data that is not a must for the execution of a user program), so that the CUP file that is downloaded to the controller has minimal size.

Designed for easy expansion

The User-Program implementation in the controller (and at the PC Suite) is designed in a way that it can be easily expanded at any time in the future. For example, it is very easy to add a math function to the list of supported functions. However, this must be done by Agito-Akribis.

Related communication messages

Note:

Some of the functions below are future features. See below for details.

In order to use the User-Program support of the controller, as well as to enable proper debugging of a User-Program, the controller supports the following communication messages:

- Download User-Program
- Upload User-Program (future feature)
- Execute User-Program thread (multi-threads is a future feature)
- Halt User-Program thread (multi-threads is a future feature)
- Halt all User-Program threads (multi-threads is a future feature)
- Define thread priority (multi-threads is a future feature)
- Upload version information (future feature)
- Set/remove breakpoints (future feature)
- Execute single command
- Report User-Program status per thread (future feature)
- Report User-Program run-time error per thread
- Set/Report User-Program program location per thread (future feature)
- And more … (see detailed list later on within this document).

Expanding the controller communication language syntax

This chapter describes the differences in the communication language syntax between an Agito-Akribis controller without the user program feature and an Agito-Akribis controller that supports user programming.

The controller communication language syntax is expanded in order to support the control, execution and monitoring of user programs by the controller.

The additional features are:

- New keywords:

  Many new keywords are added to the controller to support the user program feature. All these new keywords are listed within this document.

  For example:

ProgDownload
ProgRun
Jump
Call
Return
Math

And many more …

- Command keywords supports array indexing:

  Command keyword can be now defined as arrays with given index range. In case a command is defined as "arrayed command", it must appear with [index value] and the index value must be in the defined range.

  These tests are handled by the interpreter similarly to arrayed parameter.

  If all is OK, the index value is used by the command function as defined for each specific command.

  For the user program execution related commands, it will be used (in most vases) to indicate the addressed user program thread. In this way, we can define the number of multiple threads supported by the controller, independently of the number of axes supported (indicated by the first letter of a message).

  For example:

ProgHalt[2]	means to halt the execution of user program thread 2

  Similarly, for some of the Function keywords (like Jump, Math), the index of the array will be used to define the operation of the function.

  For example:

Math[1]	performs Addition.
Math[2]	performs multiplication
Math[30]	performs Sin()
Jump[1], location	jumps unconditionally to the location
Jump[2], location	jumps to location only if the value in stack is positive.

  Parameters that refer to threads will also use array indexing, but here there is nothing new in the communication language structure.

  For example:

ProgStat[3]	inquires the status of user program thread 3

- Command keywords now support optional argument:

In addition (and independently) to the above, Command keywords can now (optionally) receive a numeric argument, such as:

ProgRun[1], 2 means to run user program thread 1 from the start point of Task 2.

Each command has a built-in definition (in the controller) if it can accept an argument and what is the range of the argument.

If it can't, and an argument is provided, it is an error.
If it can accept an argument, the argument is a must.

If all is Ok, the argument value is provided to the function that uses this information as defined for each specific command.

As shown above, the user program related command keywords will use this argument feature to indicate, for example, a Task number, or a Function number.

The Return function, for example, uses this argument feature to optionally return a value from a function.

- Additional attributes:

  Additional attributes shall be added for each keyword, such as:

  (Some of the keywords that are used for following examples are new keywords that are defined later within this document).

  Not allowed over the communication lines (for example: "Jump", "Math")
  Not allowed from a user program (for example "DownloadFW", "ProgDownload")
  Not allowed when there is no user program in the controller (for example: "ProgRun")
  And more …

- Additional error codes:

  Additional error codes were added to the interpreter, such as:

  No user program in the controller.
  User program is running.
  Specified user program thread is already running.
  Specified user program thread is not running.
  Command argument has wrong format
  Command argument out of range
  Command does not support argument
  And more …

- Implied references to user program thread:

Some of the new keywords that relate to use program control and monitoring are defined as arrays, to provide access to each specific user program thread. These are mainly the messages that are expected over the communication channels, such as:

ProgRun[ThreadNumber],
ProgHalt[ThreadNumber],
ProgStat[ThreadNumber].

These keywords can be used also within a user program, but they must have thread number indication as the array index, to indicate to which thread they refer.

However, many of the new keywords (mostly those that refer to calculation of expressions and flow control, such as: Jump, Compare, Call, Return, Math, PushParam…) are not defined as arrays (or defined, but for other reasons) and they include no reference to a given thread. Why?

These keywords have implied reference to a thread. They are part of a user program code and when executed, they are executed as part of a given thread. As a result, naturally they refer to the thread that is executing them! (it can be different thread number at different times).

When thread 3, for example, is executed, and it reaches the following message (for example):

AMath[…]

As explained above, the axis letter is ignored (but must be a valid letter for the controller). Clearly, the "Math" function shall perform the math operation over the expressions stack (defined in details below) of thread number 3, although this is not specifically indicated.

If the same message will be reached by, for example, thread 1, it shall "Math" the values from the expressions stack of thread1.

And what if such a message (like Math) is used over one of the communication lines?

In such cases, the controller is using an extra thread, that is not normally accessible as part of a user program. Accessing such keywords over the communication is meaningless beside for training and debugging.

- Source of an executed message is a user-program:

  Until now, a message can have one of the following sources: RS-232 port, CAN port (with indication of the mailbox) or internal.

  The controller response to the input message is a function of the source of the message. For RS-232 and/or CAN, the response message is sent to the same communication channel from which the message arrived. In parallel, ErrLog[] is updated in case of any error.

For internal source, no response is sent and the ErrLog[] is updated in case of errors.

Now the controller supports an additional source for a message: The user program (with indication of the thread). In this case, no response message is sent, the relevant expressions stack may be updated and the ProgError is used to report run-time errors, which are also reported at the ErrLog[].

High level Programmer User-Program language (PUP)

Users write user-programs using the Programmer User-Program (PUP) language.

This language consists of the following groups of language statements:

- Comments.
- Compiler directives.
- Flow control.
- Expressions, operators and math functions.
- Tasks and Functions.
- User program and threads execution control and monitor.
- Any standard message as supported over the communication channels.
- Any statement of the low level Controller User Program (CUP, see below).

The sections below list the available language statements for each of these groups, as well as their format/syntax (briefly).

Some general definitions of the PUP language are first required:

General definitions of the PUP language syntax

The following are general definitions for the PUP language:

- Tabs are ignored.
- Blanks at the beginning of each line are ignored.
- Multiple blanks within a line are considered as a single blank.
- Single blanks are used only as separators between tokens.
- Empty lines are ignored.
- End of line can be one of: CR+LF, or LF, or CR.
- The language is case sensitive.
- Each line may contain no more than one statement (which can be appended with a comment at the end of the line).
- A statement must be contained within one line (no continuations of lines).

Comments

The following table presents the means to include comments within the user program:

Statement	Description
// ….	At the beginning of a line indicates that this line is a comment and is ignored by the compiler
…. // …	Within a line it means that the rest of the line is a comment and is ignored by the compiler

Compiler directives

The following table presents the supported compiler directives:

Statement	Description
#include FileName #include “FileName”	The content of the file FileName is inserted as it is instead of the #include directive. FileName can be of type *.puh only. FileName can include a path. If path is not provided, the compiler will look for it at the same directory where the user program file is located.
#define String1 String2	Following this directive, whenever the compiler encounters String1 within a given line, it will replace it by String2 before compiling this line
#definevar VariableName AGenData[N]	Following this directive, the VariableName string can be used as a name of a variable. AGenData[N] will be used as a place holder for this variable (integer). N shall be a number within the range of the GenData[] array in the controller.
#definevar ArrayName[ArraySize] GenData[N]	As above, but this statement defines an array. AGenData[N] till AGenData[N+ArraySize-1] will be used as place holders for the array. Arrays are indexed starting at Index of 1.

Statement	Description
#information AnyString	All the AnyStrings of all the #information directives are downloaded to the controller with the compiled program. The user can then request, over the communication, to read these strings. It can be used by the programmer to inform the user about the version of the program, about its status, about the status of a given function within the program etc.
#event EventID FunctionName **(Note: this directive is a future feature)**	Links a given function to one of the events that are supported by the controller (identified by the value of EventID). Only one function can be linked to any given EventID.

Directives can be located at any anywhere in the user program.

Flow control

The following table presents the language statements that enable program flow control:

Statement	Description

if (LogicalExpression)
else if (LogicalExpression)
else
end
	Similar to the C language "if" statement.

while (LogicalExpression) continue break end	Similar to the C language "while" statement.
for (InitialExpression, LogicalExpression, LoopExpression) continue break end	Similar to the C language "for" statement.
switch(Expression) case value break default end	Similar to the C language "switch" statement.

Statement	Description
(note: switch is a future feature)	
GoTo, TaskNumber (note: GoTo is a future feature)	The user program location pointer will be updated to the location of the first statement within the specified Task. **This statement shall be used only when must, as using jumps is not recommended in structural programing**.
ProgFuncCall, FunctionNumber Return	Calls a function. Upon "Return" from that function, the execution will continue at the next statement.

Where relevant, nesting of flow control statements is supported up to controller dependent depth for each statement type.

See below definitions of Expression and LogicalExpression, as well as for TaskNumber and FunctionNumber.

Expressions, operators and math functions

The following table presents how the PUP language supports expressions, operators and built-in math functions:

Statement	Description
MathFunctionName (Expression1, Expression2, …)	MathFunctionName can be one of: log, log10, exp, sin, cos, tan, asin, acos, atan, atan2, power, abs, sqrt And the list can be easily expanded as may be required. Each function has a number of required input arguments and the statement must include this number of input arguments. For trigonometric functions, angles and fractional returned values are represented in a controller dependent way.

Statement	Description
Expression	Any combination of operands (variables or Expressions by themselves) and operators. See below list of available operators.
LogicalExpression	Any combination of operands (variables or expressions or Logical Expressions) and logical operators. See below list of available logical operators.
Variable = Expression Variable = LogicalExpression	Assignment to a variable. Variable can be any user defined variable (see #definevar above) or any parameter of the controller.
Numbers (constants)	Numbers are in decimal format by default. 0x indicates number in HEX format 0b indicates number in binary format Leading zeros (after the 0x,0b) are ignored. (Note: Ox and 0b formats are future feature)

Available operators for expressions

The following is a list of supported operators:

- Parentheses: ( )
- Multiply, divide and module: * / %
- Add and subtract: + -
- Bitwise AND: &
- Bitwise OR and XOR | ^
- Shift left and right (arithmetic): << >> (note: future feature)

Also supported are the following unary operators (act on a single operand):

- Negate: - (if located in the left side of a single operand)
- Bitwise NOT (complement): ~

Available operators for logical expressions

The following is a list of supported logical operators:

- Brackets: ( )
- Logical AND: && (similar implementation to C)
- Logical OR: || (similar implementation to C)
- Comparisons: < > <= >= == !=

Also supported are the following unary logical operators (act on a single operand):

- Logical NOT: ! (if located in the left side of a single operand)

Operators precedence

The operators within an expression and/or a logical expression will be executed according to the following precedence (starting from the highest precedence). Operators that appear under the same bullet have identical precedence and will be executed from left to write (within the expression):

- ( )
- !, - (unary)
- *, /, %
- +, -
- <<,>>
- <, >, <=, >=
- ==, !=
- &
- ^
- |
- &&
- ||
- = (assignment)

In case of complex expressions, we strongly recommend using brackets to enforce the desired precedence of the expression evaluation. This will have no effect on the expression execution time at the controller.

Examples for Expressions and Logical Expressions

Variable1 = Variable2 * 35 + Variable3

ASpeed = AAInPort * AAInFactor + AAInOffset

If ((ADinPort & DigitalInputsMask) == 0x0034)

Tasks and Functions

The following table presents the language statements that relate to tasks and functions.

Note that Tasks and Functions can be included only within a PUP file. The resulted conclusion is that a PUH file may include only comments and/or compiler's directives (non-executable statements).

Task is a segment of code that you can execute (by the ProgRun keyword for example). The program does not return from a Task (it can be executed for ever in a loop, or be halted after some time) and you can't Call() a Task. Functions, on the other hand, are to be Call()'ed and must be Return'ed. However, Functions cannot be executed in any other but calling them (from within another function of a task).

A function can be an Event Function and in such case, if Thread 1 is running (main thread) and if the event is triggered, the user program engine will automatically call this function (like an interrupt).

Statement	Description
AProgTask[TaskNumber] And typically at the end of the task: AProgHalt[TaskNumber]	A definition of the starting point of a program segment (or module) that generally performs a given task (like: Homing). The TaskNumber is limited to a value that is controller dependent. This line is not an executable line and just indicates the location of the Task (which is the first executable line following this line. A task is executed using: AProgRun[ThreadNumber], TaskNumber Good programming practice is to place one task within each PUP file and to name the PUP file at least similarly (if not identically) to the task. Task must be followed by a AProgHalt[] so that the controller will not continue to execute the next lines of code. AProgHalt[] can be avoided only if the task is an endless while()

Statement	Description
AProgFunc[FunctionNumber], EventNumber Return or AReturn	A definition of the starting point of a program segment (or module) that is a function that shall be called to perform a specific functionality. Upon completion, the program execution continues at the next line following the call to the function. If EventNumber is 0 (or omitted at all), this is a normal function that is not tied to an event. If EventNumber is not 0 (and within the range of number of supported events), this function is tied to this event and it will be automatically called when the event is triggered. A function that is tied to an event can be also called normally. It is not allowed to tie more than one function to a given EventNumber. The FunctionNumber is limited to a value that is controller dependent. This line is not an executable line and just indicates the location of the Function (which is the first executable line following this line. A function is called using: AProgFuncCall, FunctionNumber A function must include at least one “Return” statement. When returning from a function that was called as an event function, this event will be flagged as it is ready again for triggering. Practically, a function shall not include an endless loop.
Function that return a value. **(Note: this is a future feature)**	TBD

User program and threads execution control and monitor

All the statements that control the user program execution and the threads execution are available also as communication messages to the controller (as these activities shall not only be available from the user program, but also by the user over the communication lines).

As a result, please refer to the Chapter "User-Program related communication messages" later on within this document for the detailed list of these statements.

Any message as supported over the communication channels

As the user program is actually a way to define a script of communication messages to be executed internally (and autonomously) by the controller, clearly any message that is supported over the communication lines (ASCII syntax) is a basic statement within the user program.

For example:

```
ASpeed = 100000
AmotionMode = 2
ABegin
```

Note that a simple assignment statement (constant numeric value assignment) is considered as a built-in message to the controller and is not compiled. However, once the assignment is more complex (right side is not a constant numeric value), the compiler will treat the statement as an expression and will compile it accordingly (convert it into the low level Controller User Program language).

Any statement of the low level Controller User Program (CUP)

Statements of the low level Controller User Program language (CUP, see below) can be used as a statement within a Programmer User Program.

Low level Controller User-Program language (CUP)

General

The PC Suite compiles the user program as written by the programmer (in high level Program User Program – PUP – language) into a low level Controller User program – CUP – language, which is later on, upon successful compilation, downloaded to the controller.

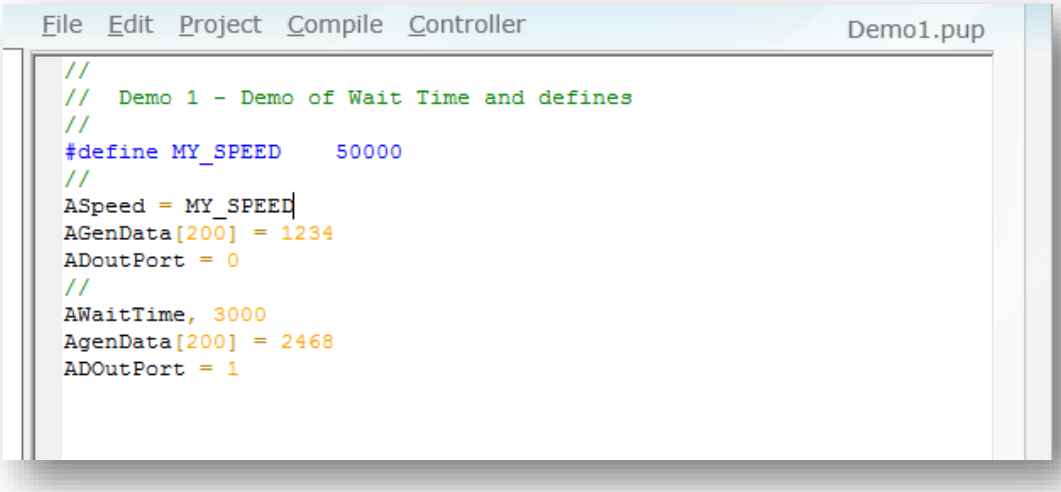
The PC Suite's compiler collect and convert the full set of PUP and PUH files that are included within the compiled project, to create a single CUP file that can be download to and executed by the controller.

The CUP file includes the PUP user program, converted to the CUP language, and some additional information as will be described later on within this chapter.

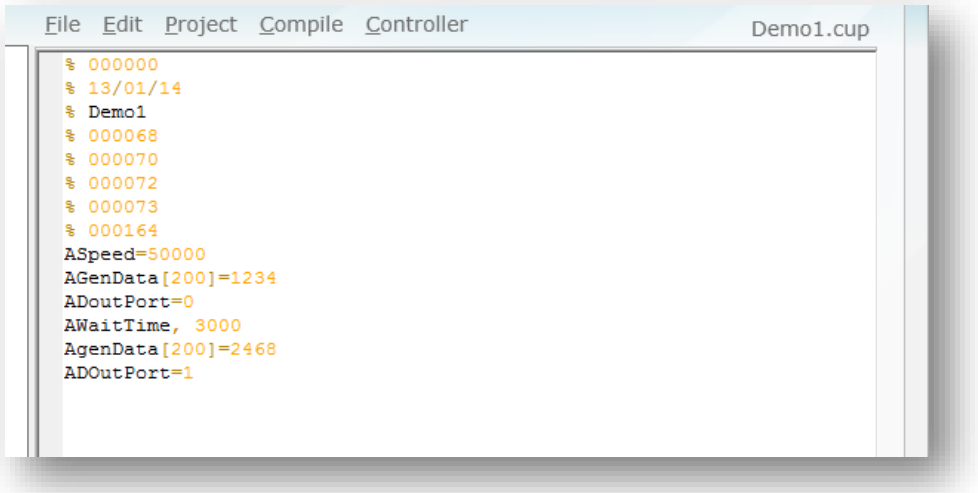
Practically, most of the executable statements within a PUP files are not modified at all and are copied as is (or almost as is) to the CUP file.

For example:

A section of a PUP file:



Is converted to the following section of CUP file (comments are removed, definitions are replaced and spaces are removed):



(For now, please ignore the header of the file, it will be later explained)
This looks just like a batch of messages over the communication line.

However, some parts of a PUP file (flow control, expressions …) must be converted into lower level set of keywords, as supported by the controller language. This is described in details later within this chapter.

Relationships between a PUP file and a CUP file

The PC Suite, during compilation of a project (a set of PUP and PUH files), creates a table with relevant information at the beginning of the CUP file. This information can be used by the controller to properly access the various parts of the program, as well as to hold information regarding this program.

Such linkage is required for:

- Line numbers (program location pointer).
- Tasks and functions location.
- Range of low level program that relate to any given high level statement.
- Comments.
- Definitions and variable naming.
- And similarly.

General definitions of the CUP language syntax

The following are general definitions for the CUP language:

- A CUP file starts with a table of information as created by the PC Suite during compilation. See details later on within this chapter.

- The rest of the file contains the CUP program, according to the following syntax rules:
  - No tabs are allowed.
  - No blanks are allowed (except in some cases as part of the CUP file header, see below).
  - No empty lines.
  - End of line will be always CR.
  - Case sensitivity: Preferably using first letter of each word capital, such as: AProgRun (from the controller point of view, there is no case sensitivity).

How PUP statements are implemented in CUP file

The following sections describe the relevant controller language keywords that are used to handle each group of the high level statements.

Please note that these keywords are accessible only from a user program (trying to send one of these low level keywords over the communication with result with an error).

Comments

Comments are not copied from a PUP file to a CUP file. A CUP file contains no comments.

This is to minimize the size of a CUP file, as it is downloaded to the controller, where FLASH memory size is "expensive".

Compiler directives

The following table presents the implementation of the compiler directives at the CUP file:

Statement	Description
#include FileName	Handled by the PC Suite. Not included within a CUP file.
#define String1 String2	As above
#definevar VariableName GenData[N]	As above
#define ArrayName[ArraySize] GenData[N]	As above

Statement	Description
#information AnyString	The compiler creates a table with all the "AnyString"'s and include it in the table of information that it creates at the beginning of the CUP file. Spaces are allowed in the CUP file only as part of these strings. A dedicated controller message (ProgInfo, see next chapter) can be used to inquire this list of strings over the communication lines. Clearly, the #information line is not included as-is in the CUP file.
#event EventID FunctionName (**note: this directive is a future feature**)	The compiler creates a table of pointers (to functions in the CUP file) for the supported events and includes it in the table of information that it creates at the beginning of the CUP file. This table is read by the controller upon successful download process and is used for proper execution of the event triggering when a user program is executed. Clearly, the #event line is not included as-is in the CUP file.

Flow control

The flow control statements (at the PUP file) are complex statements that are replaced by a sequence of low level commands (at the CUP file).

The following controller keywords are used by the compiler to implement the PUP flow control statements in CUP language.

Note the usage of the notation "Expressions Stack" in the following table. The meaning of this stack will be explained in detailed within the next section.

Note:

The table below shows a list of keywords, like: JumpEQ, JumpGT, CompareNZ and so on. Actually, the controller supports only two basic keywords: Jump and Compare.

A typical Jump message looks like:

AJump[3], 4567

The value of 3 represents the type of jump to perform (unconditional, EQ, NZ, GE, GT, …).

The same is applicable for Compare.

Please refer to the Communication Keywords Reference Manual for detailed list of Jump types and Compare types (under the dedicated page for each of these keywords).

Keyword	Description
Jump, DestinationPointer	Change the program location pointer to be equal to the DestinationPointer
JumpEQ, DestinationPointer	As above, but only if the 2nd and the 1st elements in the Expressions Stack are equal. Remove these two elements from the stack.
JumpGT, DestinationPointer	As above, but only if the 2nd element is greater than 1st element
JumpGE, DestinationPointer	As above, but greater than or equal to
JumpLT, DestinationPointer	As above, but less than
JumpLE, DestinationPointer	As above, but less than or equal to
JumpNE, DestinationPointer	As above, but not equal
JumpZ, DestinationPointer	As above, but only of the 1st element is zero. Remove this element from the stack.
JumpNZ, DestinationPointer	As above, but only if it is not zero
CompareEQ	Compare the 2nd and the 1st elements in the Expression Stack and replace both of them with 1 if they are equal or 0 if they are not equal.

Keyword	Description
CompareGT	As above, but greater than.
CompareGE	As above, but greater than or equal to
CompareLT	As above, but less than
CompareLE	As above, but less than or equal to
CompareNE	As above, but not equal
CompareZ	As above, but only the 1st element is compared to zero
CompareNZ	As above, but only if not equal to zero
ProgFuncCall, FunctionPointer	Calls to a function. Push the location pointer of the next executable message in the CUP program into the Calls Stack and change the program location pointer to be equal to the FunctionPointer
Return Return with a return value will be also supported in the future.	Returns from a function. Push the ReturnValue to the Expressions Stack and Jump back to the call location by popping a location pointer from the Calls Stack
WaitStatus[StatusType],StatusValue	Waits for a specified status (StatusType) to become true, using also the StatusValue. Can be to wait for end of motion, or a value of a user flag, etc. Refer to the Communication Keywords Reference Manual for detailed list of the supported status types (under the dedicated page for the WaitStatus keyword).
GetStatus[StatusType] (note: this keyword will be supported in the future)	Gets the value of a specified status (StatusType). It is a read-only parameter keyword. StatusType is equivalent to the types used by WaitStatus, see above.

In addition to these controller keywords, the compiler needs also to handle the Expressions and LogicalExpressions that are included within the flow control statements. Refer to the next section for details about the implementations of these expressions in CUP language.

The above controller keywords are optimally used (considering execution speed at the controller) to create CUP code that will replace all the following PUP flow control statements:

If, while, for, call and their related statements: else if, else, continue, break …

Expressions, operators and math functions

Expressions are complex statements of the PUP language. The controller, using its CUP language does not support expressions but only unary and binary operators. As a result, the PC Suite compiler, translate any expression into a set of calculations that can be written in CUP language.

Using this method, the controller interpreter is kept simple, small and fast and the user program execution speed is optimal (the controller shall not evaluate and interpret complex expressions, precedence etc.).

In order to support the implementation of expressions, the CUP language supports an Expressions Stack, dedicated keywords to push/pop to/from this stack and a set of keywords to cover all required operators, as listed below.

Implementation of any expressions is converted to a set of push operations, operators and pop operation for the assignment.

An example may be the best way to show this method:

Assume the following expression in a PUP language:

ASpeed = AAInPort * 100 + (AGenData[200] + 100) / 30

It will be converted into the following set of CUP messages (expression stack content shown in brackets):

	(Empty)
PushParam, AAInPort	(AAInPort value)
APushConstant, 100	(100, AAInPort value)
AMath[MULTIPLY]	(AAInPort*100)
APushParam, AGenData[200]	(AGenData[200], AAInPort*100)
APushConstant, 100	(100, AGenData[200], AAInPort*100)
AMath[ADD]	(AGenData[200]+100, AAInPort*100)
APushConstant, 30	(30, AGenData[200]+100, AAInPort*100)
AMath[DIVIDE]	((AGenData[200]+100)/30, AAInPort*100)
AMath[ADD]	(AAInPort*100+(AGenData[200]+100)/30)
APopParam, ASpeed	(Empty)

At the end of this sequence of low level messages, ASpeed gets the desired value of the expression.

Note that PushParam is a new keyword that gets the "name" of a parameter keyword and pushes its current value to the expressions stack and PopParam means to pop a value from the expressions stack into the specified parameter.

Note that the actual CUP file language does not support a name of a parameter as a part of the message. It actually contains a numeric value that points to this parameter. This numeric value is automatically calculated by the PC Suite compiler. Please contact Agito-Akribis in case you are interested in a detailed description about how this numeric value is calculated.

The expressions stack is always kept balanced and is usually empty. Of course, the controller checks and generate errors if trying to push into a full expressions stack or trying to pop from an empty stack, or if the expressions stack has not enough elements for a given operator or keyword. Such error will halt the user program and will set a status of run-time error.

The Expressions Stack has a depth of 50 elements (per each program thread). This shall enable very complex expressions. Of course, the stack is cleared upon downloading of a new user program or upon resetting the user program.

Note – Order of items in the expression stack:

The 1st element in the stack is defined as the 1st one to be popped (last pushed element), 2nd element is the 2nd one to be popped and so on. This means that the notation "1st" refers to the latest pushed item, which is the first item to be popped out (LIFO – Last In, First Out stack).

Example:

Starting with an empty stack		(Empty)
APushConstant, 100	→	(100) 100 is now the 1st element in the stack
APushConstant, 200	→	(200, 100) 200 is now the 1st element in the stack 100 is now the 2nd element in the stack
APushConstant, 300	→	(300, 200, 100) 300 is now the 1st element in the stack 200 is now the 2nd element in the stack 100 is now the 3rd element in the stack

When reading the stack (using ProgExpStack[] keyword), the value of 300 will be found at location 1, and the value of 200 will be found at location 2 and so on.

When popping values from the stack (using the PopParam keyword, or implicitly using Math or Jump) the 1st element in the stack will be popped first. After the 1st element has been popped out, the 2nd element becomes the 1st, and so on…

The following controller keywords are used by the compiler to implement expressions, logical expressions and math functions:

Keyword	Description
PushParam, <Complex CAN Code>	Push the current value of the parameter which is pointed by the "Complex CAN Code", into the expression stack. Error is created if the stack is full.
PushConstant, <number>	Push the constant value into the expressions stack[1]
PopParam, <Complex CAN Code>	Pop a value from the expression stack and assign it to the parameter[2]
Math[LOG[3]]	Pop a value from the expressions stack, perform log(value) to this value and push the result to the stack
Math[LOG10]	As above but Log10(value)
Math[EXP]	As above but evalue
Math[SIN]	As above but sin(value)[4]
Math[COS]	cos(value)
Math[TAN]	tan(value)
Math[ASIN]	asin((value)
Math[ACOS]	acos(value)
Math[ATAN]	atan(value)
Math[ATAN2]	Takes two values from the expressions stack, performs atan2(2nd value, 1st value) and push the result to the stack
Math[POWER]	As above, but performs (2nd value)(1st value)
Math[ABS]	abs(value)
Math[SQRT]	sqrt(value)

[1] Of course, error is created if the stack is full.
[2] Of course, just as for communication message, an error will be created if the assignment is not allowed (read only parameter, not allowed during motion, value out of range etc.). And, of course, also if the stack is empty.
[3] LOG (as LOG10 and others below) indicates a constant value. The PUP file can even use #define to indeed use LOG and not a numeric value. Refer to the dedicated page of the Math keywords, at the Communication Keywords Reference Manual for a detailed list of the supported Math types.
[4] All trigonometric functions assume angles in radians

Keyword	Description
Math[MULTIPLY]	Takes two values from the expressions stack, multiply them and push the result to the stack
Math[DIVIDE]	As above, but: (2^{nd} value) / (1^{st} value)
Math[MODULO]	As above, but modulo
Math[ADD]	As above, but add the two values
Math[SUBTRACT]	As above, but subtract
Math[NEGATE]	Pop a value from the expressions stack, negate it and push it back
Math[BITWISE_AND]	Takes two values from the expressions stack, perform bitwise AND between them and push the result to the stack
Math[BITWISE_OR]	As above, but bitwise OR
Math[BITWISE_XOR]	As above, but bitwise XOR
Math[BITWISE_NOT]	Pop a value from the expressions stack, bitwise NOT it and push it back
Math[LOGICAL_AND]	Takes two values from the expressions stack, perform logical AND between them and push the result (0 or 1) to the stack
Math[LOGICAL_OR]	As above, but logical OR
Math[LOGICAL_NOT]	As above, but logical NOT

In addition, the implementation of logical expressions uses the comparison keywords as listed in the previous section.

Using the above defined controller keywords, as well as the expressions stack; the compiler can generate code in CUP language for any complex expression or logical expression in PUP language, while taking care for the calculation precedence.

Tasks and Functions

In order to support calls for functions and returning from a function, the controller supports the following keywords, which were already listed above as part of the flow control section:

ProgCallFunc, FunctionNumber
Return

A function is defined using the:

ProgFunc[FunctionNumber], EventNumber (0 for normal function, or: avoid the ", EventNumber")

Statement.

Tasks are defined using the:

ProgTask[TaskNumber]

Keyword, and are executed using the keyword:

ProgRun[ThreadNumber], TaskNumber

The information table at the beginning of a CUP file

The PC Suite compiler generates a CUP file that is downloaded to the controller. The CUP file includes two sections.

The first section is an area where the compiler place tables of information as may be required by the controller to optimally perform the user program execution and related operations.

The second section is the user program itself, in CUP language.

This section lists the contents of the first section of a CUP file (please address Agito-Akribis in case that a more detailed description is required):

- CRC value for the overall file.
- Date of download.
- Name of the download CUP file at the PC (can include spaces).
- Pointer to the table of information data.
- Pointer to the table of Tasks and functions (maybe two separated tables and two separated pointers, as Tasks and Functions are not to be handled the same (Tasks can be executed, Functions can be only called). Actually, we need only table of Tasks!!!
- Pointer to the table of events assignments to functions.
- Pointer to the first character of the user program.
- Pointer to the last character of the user program (also the length of the CUP file).
- A table holding all the strings defined by the programmer using the #information compiler directive. This table is uploaded over the communication (together with the listed above CRC, date of download and file name), as a response to the ProgInfo command keyword.
- A table listing all the Tasks as defined in the PUP files, and pointer, for each Task, into the relevant location in the CUP file.
- A table which lists type of events and the function that is related to this event.

Note that the overall size of a CUP file (the information section and the program itself) is limited by the size of the FLASH sector(s) that is allocated at the specific controller for the user program.

More about Event Functions

Some clarifications and additional information about Event Functions handling:

- If Event Function is triggered while Thread 1 is running but is handling a Wait message (such as: WaitTime), the Event Function will be called (immediately) and when completed, the Wait message will be re-executed (WaitTime, for example, will start waiting the whole specified wait time).
- The controller supports up to 5 events (EventNumber = 1 to 5). Event number 1 is a fast event, as defined below, while events 2 to 5 are normal events.

  Note: different number of events can be supported depending on the product and the FW version.

- Fast event: The detection of the event trigger itself is performed each control interrupt, meaning at resolution of 61 micro seconds.
- Normal event: The detection of the event trigger itself is performed within the control interrupt, but with a resolution of 1msec.
- If multiple threads are executed (and assuming Thread 1 is one of them), the respond to a detected event trigger is performed only when the user program execution engine reaches the execution of Thread 1. This may create some additional delay if many threads are running together.
- The ProgEventType defines the type of the trigger and supports the following values:

ProgEventType value	Type of trigger
1	Greater than (>)
2	Equal (==)
3	Not equal (!=)
4	Less than (<)
5	Rising edge
6	Falling edge
7	Manual → Not supported. Will not generate a trigger.
8	On change

- The "On Change" triggering type is an interesting mode. The trigger is detected when the value of the trigger parameter is different from its initial value. The initial value is updated upon few cases as listed below. One of them is upon returning from servicing

this event. This mean that it can be used for re-servicing the event whenever the trigger parameter is being changed…

- Few events can use the same value at ProgEventPar, meaning to be triggered using the same controller parameter (with the same or different trigger type, value and mask).
- Understanding the internal logic of event triggering is a must to properly control and predict the servicing of the event functions. The following items and table present this internal logic:

  - The Built-in logic holds the following variables, per each event (not directly accessible by the user):

    ProgEventParInitialVal
    ProgEventParPrevVal

  - ProgEventParInitialVal holds the initial value of the trigger parameter and is used for the "On Change" trigger type.
  - ProgEventParPrevVal holds the previous value of the trigger parameter and is used for the "Rising Edge" and "Falling Edge" trigger types.
  - The following table shows when these variables are updated.

Upon..	Internal logic
Power On or Reset	ProgEventOn is 0 ProgEventGEn is 0 ProgEventEn is 0 for all events ProgEventStat is 0 for all events ProgEventPar, ProgEventType, ProgEventMask and ProgEventVal are loaded from the Flash, for all events. ProgEventParInitialVal and ProgEventParPrevVal are set to the current value of the event trigger parameter, per each event.

Load from Flash	ProgEventPar, ProgEventType, ProgEventMask and ProgEventVal are loaded from the Flash, for all events. ProgEventParInitialVal and ProgEventParPrevVal are set to the current value of the event trigger parameter, per each event. ProgEventOn, ProgEventGEn, ProgEventEn (for all events) and ProgEventStat (for all events) remain unchanged.
Download User Program	ProgEventOn, ProgEventGEn, ProgEventEn (for all events) and ProgEventStat (for all events) are set to 0. All other parameters and internal variables remain unchanged.
Reset All Threads	ProgEventOn, ProgEventGEn, ProgEventEn (for all events) and ProgEventStat (for all events) are set to 0. All other parameters and internal variables remain unchanged.
Reset Thread 1	ProgEventOn, ProgEventGEn, ProgEventEn (for all events) and ProgEventStat (for all events) are set to 0. All other parameters and internal variables remain unchanged.
Reset any Thread except thread 1	Nothing
Start servicing an event	The ProgEventStat of this event is set to 2 ("in service" and blocking triggering of this event).
Return from servicing an event	The ProgEventStat of this event is set to 0 ("waiting for trigger"). ProgEventParInitialVal and ProgEventParPrevVal are set to the current value of the event trigger parameter, for this event.
Upon checking for trigger detection at the control interrupt (once per 61µs per event 1 and 1ms per events 2-5)	ProgEventParPrevVal is set to the current value of the event trigger parameter, for this event, after checking for the trigger condition.
Assigning a value to: ProgEventPar[EventNumber]	ProgEventParInitialVal and ProgEventParPrevVal are set to the current value of the event trigger parameter, for this event.

Assigning a value to: ProgEventMask[EventNumber]	Nothing. However, if using the "On Change" event trigger mode, it is recommended to (re-)assign the value to ProgEventPar[EventNumber] after assigning a value to ProgEventMask[EventNumber] to ensure proper initialization of all internal variables.
Assigning a value to: ProgEventType[EventNumber]	Nothing
Assigning a value to: ProgEventVal[EventNumber]	Nothing
Assigning a value to: ProgEventEn[EventNumber]	Nothing
Assigning a 0 value to: ProgEventStat[EventNumber]	Nothing
Assigning a value to: ProgEventGEn	Nothing
Assigning a 0 value to: ProgEventOn	Events are not sensed and are not handled since now ProgEventOn is 0. The ProgEventStat is set to 0 ("waiting for trigger") for all events to clear all possibly pending events.
Assigning a 1 value to: ProgEventOn	Nothing. However, if using one of the "On Change", "Rising Edge" or "Falling Edge" event trigger modes, it is recommended to (re-)assign the value to ProgEventPar[EventNumber] before Setting ProgEventOn=1, to ensure proper initialization of all internal variables.

User-Program related communication messages

To control and monitor the execution of user program threads, the set of communication messages (keywords) was expanded to support the following user program related messages (keywords).

Note that all these communication messages, just as any communication message, can be included as part of a user program. This means that from within a given thread of a user program, it is possible to control the execution of other threads (run, halt, get status and set priority …).

The following table presents the list of keywords that relate to user program control and monitoring:

Keyword	Description
ProgRun[ThreadNumber], TaskNumber	Start/Continue program execution. If TaskNumber is equal to 0, the program execution continues from its current location. If TaskNumber is equal to -1, the program execution starts from the first line of the program. Otherwise, the program execution starts from the requested task.
ProgHalt[ThreadNumber]	Halt program execution of the specified thread
ProgHaltAll	Halt all the currently active user program threads
ProgPointer[ThreadNumber] (at the moment, it is a Read Only parameter)	Inquire current value of the program location pointer of the specified thread. Value of 0 is the beginning point of the program.
ProgSingle[ThreadNumber], Type	Execute single program command at the specified thread. If Type == 0, the controller will execute the current line once and may stay at this line if it is a “Wait” line and the wait condition is not yet satisfied. If Type is not equal to zero, to controller will continue to execute the current line till it will move to the next (or any other) line (Step Over).

Keyword	Description
ProgBreaks[index] = Location Index = 1, 2 or 3	Set a break point at ProgPointer=Location. Any thread that will reach this location will halt. Up to three break points are supported. The software checks from index=1 to 3. If a Location with a value of -1 is found, the checking stops. So the PC Suite must organize all break points from index = 1
ProgStat[ThreadNumber]	Inquires the status of the specified program thread: -1: No user program in the controller 0: Not running 1: Running
ProgError[ThreadNumber]	Inquires the last run-time error of the specified program thread
ProgClrExp[ThreadNumber]	Clears the expressions stack of the specified thread
ProgExpStack[ThreadNumber], Location	Inquires the program expressions stack of the specified thread, at the given location
ProgExpDepth[ThreadNumber]	Inquires the empty spaces at the program expressions stack of the specified thread
ProgClrCall[ThreadNumber]	Clears the calls stack of the specified thread
ProgCallStack[ThreadNumber], Location	Inquires the program calls stack of the specified thread, at the given location
ProgCallDepth[ThreadNumber]	Inquires the empty spaces at the program calls stack of the specified thread
ProgResetAll	Halt all user program threads and resets all its pointers and statuses
ProgEventOn	Activate ("1") or disables ("0") the handling of user program events. When disabled ("0") all pending events are cleared and events are not handled/processed at all. This includes also the sensing of events.
ProgEventPar[EventNumber]	Defines (using Complex CAN Code) which controller parameter to use for the triggering of this event. If ProgEventPar[EventNumber] is set to 0 (or to a non-valid Complex CAN code), this event will not be sensed and will not be handled.
ProgEventMask[EventNumber]	Defines a bitwise mask to apply on the user defined event trigger parameter. The mask is also applied on the trigger value.

Keyword	Description
ProgEventType[EventNumber]	Defines the type of the trigger (rising edge, equal, not equal…). Note that the four parameters to define a trigger for an event are very like the definition of a trigger for data recording.
ProgEventVal[EventNumber]	Define the value to be used for the trigger detection.
ProgEventEn[EventNumber]	Enables ("1") or disables ("0") the servicing of this event. ProgEventEn, when "0", does not disable the sensing of the event, and the event can still be sensed and possibly pending, to be serviced when enabled.
ProgEventStat[EventNumber]	Reports the state of this event. "0" for waiting for trigger, "1" for pending for service (triggered) and "2" for in service. Note that this mean that while a given event is being serviced, it can't not be triggered again, till servicing is completed (returning from the event function using the Return keyword). This parameter is R/W, so user can clear a pending event – only the value of 0 can be written to this parameter).
ProgEventGEn	Globally enables ("1") or disables ("0") the servicing of all events. ProgEventGEn, when "0", does not disable the sensing of events, and events are still sensed and possibly pending, to be serviced when enabled.
ProgPriority[ThreadNumber]=Priority	Sets/Inquires the priority of the specified program thread. Priority value is 1 to 10, where 1 is the highest priority.
ProgInfo??? (note: this is a future feature)	Inquires the list of information strings that are included with the user program: CRC value, Date, CUP File name and the text information (see the "#information" compiler directive)
DownloadUPBin	Used to download a new user program to the controller
UploadUProg (note: this is a future feature)	Used to upload the user program from the controller

Note that all the references in the above table to program location (ProgPointer, ProgBreaks) and the reference to single program command (ProgSingle) refer to the low-level Controller User Program language as saved in the controller. It is the PC Suite responsibility (using the debug information that it creates during the compilation process) to link between these low-level pointers to the high level pointers (Tasks, function names, high level program statement) as may be referred by the user.

PC Suite – User-Program development environment

Refer to the PC Suite User's Manual.