Postprocessor development

Postprocessors convert the CAM system's neutral CLData stream into the machine specific files required by a CNC control. This documentation covers the two postprocessing systems and the shared data they process.

Main sections

- SPPX postprocessors - create postprocessors in the built in Pascal like language with the Postprocessors generator.
- .NET postprocessors - build postprocessors as .NET assemblies using the DotnetPostprocessing SDK. The API reference is part of this section.
- CLData - reference for the shared toolpath and technology command stream.
- AI assisted development - configure tools and workflows for developing postprocessors with an assistant.

Choose a section above to start. The SPPX and .NET systems are independent authoring paths and use the same CLData input.

Postprocessors generator (sppx postprocessors)

The Postprocessors generator is the subsystem for building postprocessors in the built-in Pascal-like language — the `.sppx` postprocessors. This section documents the generator, its masks and registers model, and the language. External links and example resources are listed in references.

- External references and examples
- Introduction
  - The purpose of the postprocessors generator
  - The files set of the postprocessors generator
- The common organization of the work
  - The principle of postprocessor operation
  - The main window
    - The main toolbar
    - System settings
    - Editor settings
    - Machine parameters
      - Defining the data about the NC-machine and CNC-system
      - Comments transliteration
    - The block structure and format definition (Register list forming)
    - The masks for the machining commands translation
    - The programs for the CLData commands processing
    - Subprograms
    - The command processing programs compilation
    - The work with the files of technological commands
    - The test NC-code generation
    - Programs debugging
    - Postprocessor parameters inquiry while the first using
- Postprocessor masks
  - Mask structure
    - Mask element
    - Registers in the masks
    - Modifiers
    - Expressions
    - Nested mask
    - The separators of the mask elements
    - Variables assignment from the mask
    - Deferred masks
    - GMA array in masks
  - Mask management
    - Mask switches

        - The transformation of a mask to the subprogram
        - The interactive to create the masks
- Language description
    - Basic definitions
        - Conditional indications
        - Technology command programs, comments
        - Subprograms
        - The language statement conception
        - The set of symbols
        - The variables
        - Arrays
        - Mathematical expressions and functions
        - Predefined variables and functions
            - Standard mathematical functions
            - String functions
            - Miscellaneous functions and variables
            - CLData access functions and operators
        - Filter - output filter
        - Records
        - Objects
            - EulerConverter
            - Matrix
            - NCChannel
            - Point3D
            - Quaternion4d
            - StringHelper
    - Operators
        - The processing program start operator PROGRAM
        - The statement of assignment
        - The output statement PRINT
        - The input statement INPUT
            - The input from file
            - The input from keyboard
        - Conditional statement IF
        - Statement of the multiconditional execution CASE
        - JUMP statement
        - The cycle statement FOR
            - Statement CONTINUE
            - Statement BREAK
        - The cycle statement REPEAT

        - The cycle statement WHILE
        - Composite statement BEGIN...END
        - Statement to call a subprogram CALL
        - The statement of the subprogram start SUB
        - The statement of the subprogram end SUBEND
        - The statement of the procedure start PROC
        - The RETURN from a procedure statement
        - The block output statement OUTBLOCK
        - The block forming statement FORMBLOCK
        - Statement of direct output into the block OUTPUT
        - The replace statement of the substring in a string REPLACE
        - The statement to form the block by mask MASK
        - Change current NC-file operator
        - External task call operator
        - CustomData access operator CAM system
        - Operators of work with NC-subroutines
            - The operator of a output of NC-subroutine NCSUB.OUTPUT
            - NCSUB.OUTPUTALL operator
            - The operator of definition of an initial marker of NC-subroutine NCSUB.STARTLABEL
            - The operator of definition of a final marker of NC-subroutine NCSUB.ENDLABEL
            - The operator of definition of a name of NC-subroutine NCSUB.NAME
            - Technique of work with NC-subroutines
        - ReplNCStr operator to replace strings at NC-file
- AI-assisted postprocessor development
    - How the system works
    - Set up the tools
    - Organize your workspace
    - Develop with CAM Agent
    - Develop in Visual Studio Code
    - What the extensions give you
    - SPPX workflow
    - .NET workflow
    - Review and verify
    - Troubleshooting
    - Advanced setup
- CLData reference
    - CLData access functions and operators
        - Named CLData parameters
        - CLD predefined array
        - Current command (Cmd operator)

        - Browsing CLData commands
        - CLData files (CLDFile operator)
        - Project information (Project operator)
        - GMA array
    - Project information in CLData
    - Technology commands description
        - AXESBRAKE - Machine axes brakes control
        - CIRCLE - Circle movement
        - CLAMP - Fixtures clamp
        - COMMENT - Commentaries
        - COOLNT - Cooling
        - CUTCOM - Tool compensation
        - CYCLE - Hole machining cycles
            - CYCLE BORE5(209) - G85 drilling cycle
            - CYCLE BORE6(210) - G86 drilling cycle
            - CYCLE BORE7(211) - G87 drilling cycle
            - CYCLE BORE8(212) - G88 drilling cycle
            - CYCLE BORE9(213) - G89 drilling cycle
            - CYCLE BRKCHP(288) - Deep drilling with chip breaking
            - CYCLE DEEP(153) - Deep drilling with chip removing
            - CYCLE DRILL(163) - G81 drilling cycle
            - CYCLE FACE(81) - G82 drilling cycle
            - CYCLE TAP(168) - G84 drilling cycle
        - DELAY - Delay
        - EDMMOVE - EDM movement
        - EFFECTOR - End effector activation
        - EXTCYCLE - Extended cycle
            - EXTCYCLE BRKCHP - Lathe drilling with chip breaking cycle
            - EXTCYCLE DEEP - Lathe drilling with chip removing cycle
            - EXTCYCLE DRILL - Lathe drilling cycle (G81)
            - EXTCYCLE FACE - Lathe drilling cycle (G82)
            - EXTCYCLE LATHEFINISH - Lathe finish cycle (G70, G73)
            - EXTCYCLE LATHEGROOVE - Lathe grooving cycle (G74, G75)
            - EXTCYCLE LATHEROUGH - Lathe roughing cycle (G71, G72)
            - EXTCYCLE LATHETHREAD - Lathe threading cycle (G76)
            - EXTCYCLE LATHETHREADG92(404) - Lathe threading single pass cycle (G92)
            - EXTCYCLE TAP - Lathe tapping cycle (G84)
            - EXTCYCLE W5DBore5 - Drilling cycle (G85)
            - EXTCYCLE W5DBore6 - Drilling cycle (G86)
            - EXTCYCLE W5DBore7 - Drilling cycle (G87)

        - EXTCYCLE W5DBore8 - Drilling cycle (G88)
        - EXTCYCLE W5DBore9 - Drilling cycle (G89)
        - EXTCYCLE W5DChipBreaking - Drilling with chip breaking cycle (G73)
        - EXTCYCLE W5DChipRemoving - Drilling with chip removing cycle (G83)
        - EXTCYCLE W5DDrill - Drilling cycle (G81)
        - EXTCYCLE W5DFace - Drilling cycle (G82)
        - EXTCYCLE W5DHolePocketing - Hole pocketing cycle
        - EXTCYCLE W5DTap - Tapping cycle (G84)
        - EXTCYCLE W5DThreadMill - Thread milling cycle
        - EXTCYCLE WProbing - Probing cycle
            - Additional parameters
            - Boss probing parameters
            - Double wall external corner probing parameters
            - Double wall internal corner probing parameters
            - External rectangle probing parameters
            - Groove probing parameters
            - Groove probing protected parameters
            - Hole probing parameters
            - Hole protected probing parameters
            - Internal rectangle probing parameters
            - Internal rectangle probing protected parameters
            - Surface probing parameters
            - Triple wall external corner probing parameters
            - Triple wall internal corner probing parameters
            - Web probing (three points) parameters
            - Web probing parameters
    - FEDRAT - Feedrate
    - Filter - Output filter (global handler)
    - FINI - Ending record
    - FROM - Original point
    - GOHOME - Return to original position
    - GOTO - Linear transition
    - HEAD - Spindle head number
    - INSERT - Insertion
    - INTERPOLATION - Interpolation mode
    - LOADTL - Tool loading
    - MULTIARC - Multi-axis circle movement
    - MULTIGOTO - Multi-coordinate movement
    - OPSKIP - Optional skipping
    - OPSTOP - Auxiliary stop

- ORIGIN - Original coordinates
- PALETA - Palette changing
- PARTNO - Part number
- PHYSICGOTO - Physical machine axes movement
- PLANE - Working plane
- POWER - Power adjustment
- PPFUN - Postprocessor function
  - PPFUN CALLSUB(52) - NC-subprogram call command
  - PPFUN ENDSUB(51) - NC-subprogram end command
  - PPFUN ENDTECHINFO(59) - End of operation's technological information
  - PPFUN PPFUN(500) - Tool durability control command
  - PPFUN STARTSUB(50) - NC-subprogram start command
  - PPFUN TECHINFO(58) - Technology parameters command
  - PPFUN WEDMConditions(56) - EDM machining conditions specification command
  - PPFUN WEDMPassItem(57) - Wire EDM pass properties
- PPRINT - Postprocessor printing
- RAPID - Rapid feedrate
- ROTABL - Table rotation
- SAFPOS - Tool change point
- SELWORKPIECE - Active workpiece connector selection
- SINGLETHREAD - Single-pass thread
- SPINDL - Spindle
- STOP - Stop
- STRUCTURE - Toolpath structure
- SYNCAXES - Axes movement synchronization
- TAKEOVER - Workpiece takeover
- TLCONTACT - Tool contact
- WAIT - Waiting for synchronization point

External references and examples (Postprocessors generator / sppx)

External resources for authoring `.sppx` postprocessors with the Postprocessors generator.

Product documentation

Reference	URL
ENCY portal	https://encycam.com/
Postprocessor Generator user manual	Open the local guide
ENCY Academy — training courses	https://learn.encycam.com/

AI-assisted development

Download locations for the tools described in AI-assisted postprocessor development.

Reference	URL
Postprocessor tools for VS Code — releases	https://github.com/EncySoftware/postprocessor-tools/releases
ENCY documentation center	https://docs.encycam.com/portal/

See also

- .NET postprocessors — the other postprocessing subsystem, built as .NET assemblies.
- CLData — the shared input model read by both postprocessing subsystems.

Introduction

The CAM system has two independent postprocessing systems now.

1. Postprocessors generator. This is an integrated postprocessor development environment, which has its own format, programming language and debugger.
2. New open postprocessing system based on the .NET platform, using `C#` as a programming language and Visual Studio Code as an environment of development and code debugger.

Both systems are actual and can be used by your choice.

Here you will find the documentation only for the Postprocessors generator. Documentation for the second (.NET based) system is available here.

The purpose of the postprocessors generator

Postprocessors generator is an application for operating systems of the Windows family.

The purpose of the postprocessors generator is the generation of the postprocessor adjustment files to various NC-systems. These files are used by the run-time postprocessor system for the concrete NC-program generation.

It is necessary to perform the following steps to develop the postprocessor adjustment file:

- Define the data about the NC-machine and CNC-system;
- Describe the structure and the format of the block (form the list of registers);
- Design masks or programs to process technological commands;
- Save the postprocessor's tuning file.

The generation of new tuning files and the editing of existing files is allowed.

In addition to the work with data about NC-machine and CNC-system, there is the possibility of the examination of the technological commands files and the trial generation of NC-programs in the environment of the postprocessors generator.

The files set of the postprocessors generator

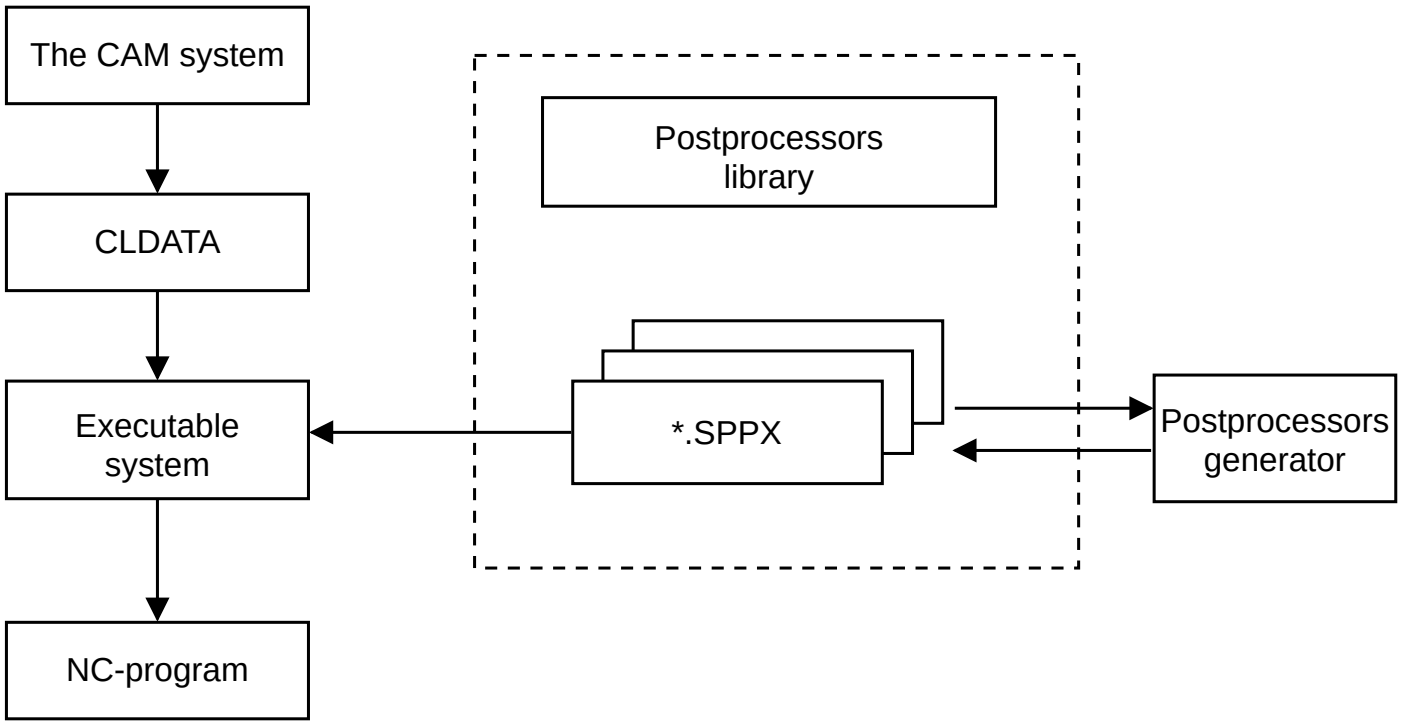
- Inp.exe – executable module of postprocessors generator.
- InpD.dll – executable system of the postprocessor.
- CLDImport.dll - CAM system projects import library.
- *.sppx – the file with a data about concrete NC-system and with the sources of programs to process technological commands.
- *.stcx – CAM system XML project files.
- *.stcp – CAM system project files.
- *.mcd – temporary files of technological commands, linked with CAM system project files.
- *.spp, *.inp – the postprocessor adjustment files for the previous version of the postprocessor generator.

The common organization of the work

No content in this page. See child topics.

The principle of postprocessor operation

Postprocessors generator allows to develop the postprocessor adjustment files for the different NC-systems (*.SPPX files). An import of the files from the previous version of postprocessor (*.SPP, *.INP files) is available also. The postprocessor adjustment file contains the descriptions of all features to generate NC-program for the defined NC-system. The executable system of postprocessor uses this description to generate the NC-programs from the files of technological commands (*.MCD, *.STCP, *.STCX files), which can be produced, in turn, by CAM system, for example.



To develop the postprocessor's tuning file, it's necessary to define the data about the NC-machine and NC-system, to describe the structure and the format of the block and to fill the masks or to design the programs to process the technological commands.

The data about the NC-machine and NC-system means the name of the NC-machine and of the NC-system, the limits for displacement along the axes and some additional data.

The structure and the format of the block are defined by the ordered sequence of the registers and by its parameters. The identifiers and the values of the registers will be output to the NC-block in the same sequence as they are placed in the list.

The mask contains the list of registers in the required order. These registers will be out to the NC-program block for the corresponding technological command.

The special problem-oriented language is used to write the programs to process the technological commands. This language allows the mathematical expressions and functions, the statements for

input/output, conditional statements, cycles, jump statement, calls of subroutines, the statements to form the NC-program blocks and the statements to work with the technological commands file.

The data definition about the NC-machine and the CNC-system, the block structure and format description, the programs design to process technological commands are performed in the postprocessors generator environment. The examination of the technological commands files and the trial generation of the NC-program are allowed in this environment too.

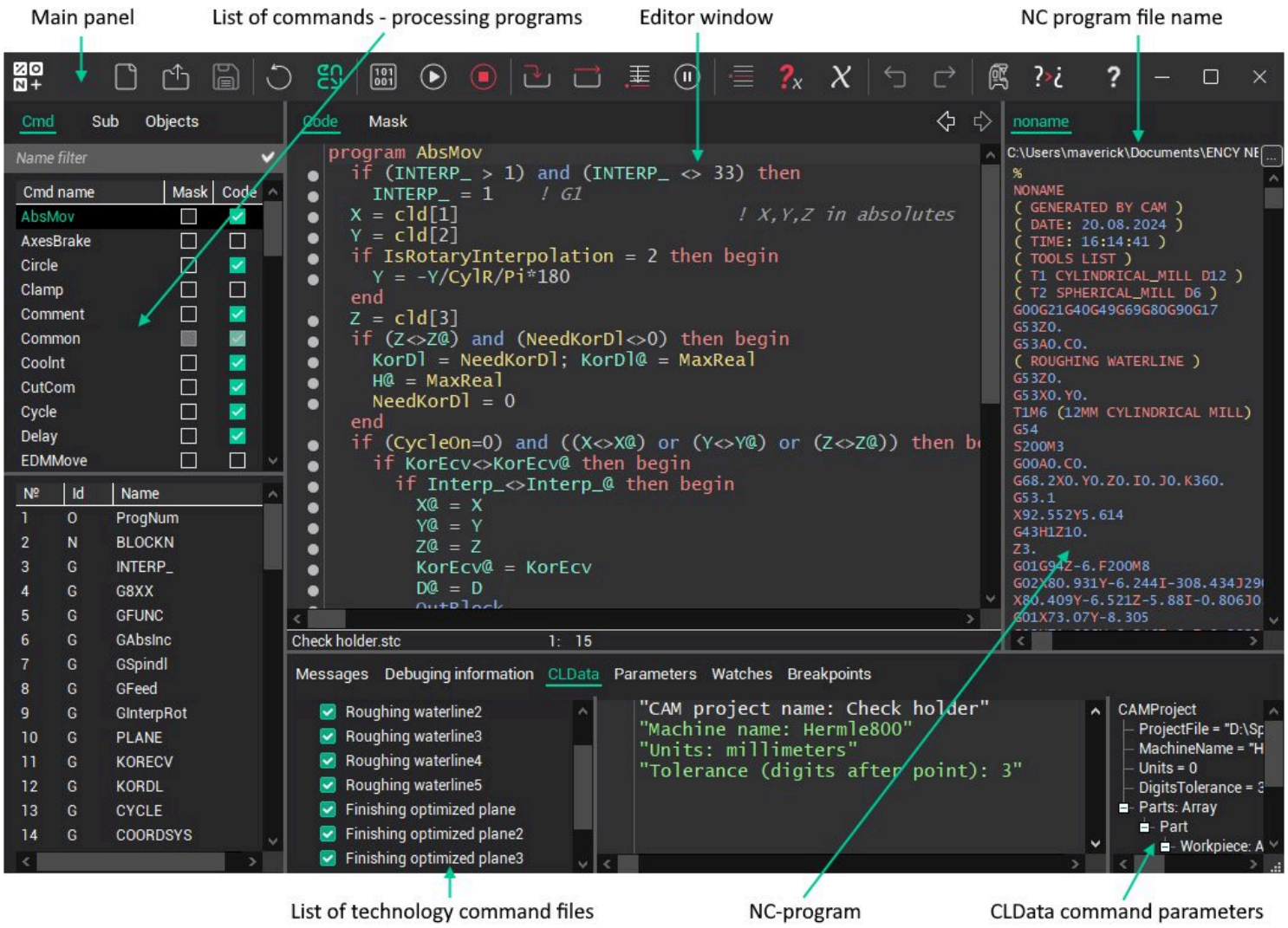
The programs sources to process technological commands, the data about the NC-machine and the CNC-system, the masks, the list and the format of registers are saved in the file with the name of concrete postprocessor and the extension *.sppx. The data from this file is used by the executable system of postprocessor to generate the NC-program for the corresponding NC-system.

The run-time postprocessor system reads the data about the machining process from the file of technological commands, analyses the code of the command and activates the program, which processes this command (the name of mask and processing program is coincident with the name of the corresponding command). Then if it was used mask, that forms line of the NC-program for corresponding of mask. The parameters of the technological command are passed via the predefined array CLD and special operator CMD. Called program can change the registers values and internal variables values and it can form the block of the NC-program.

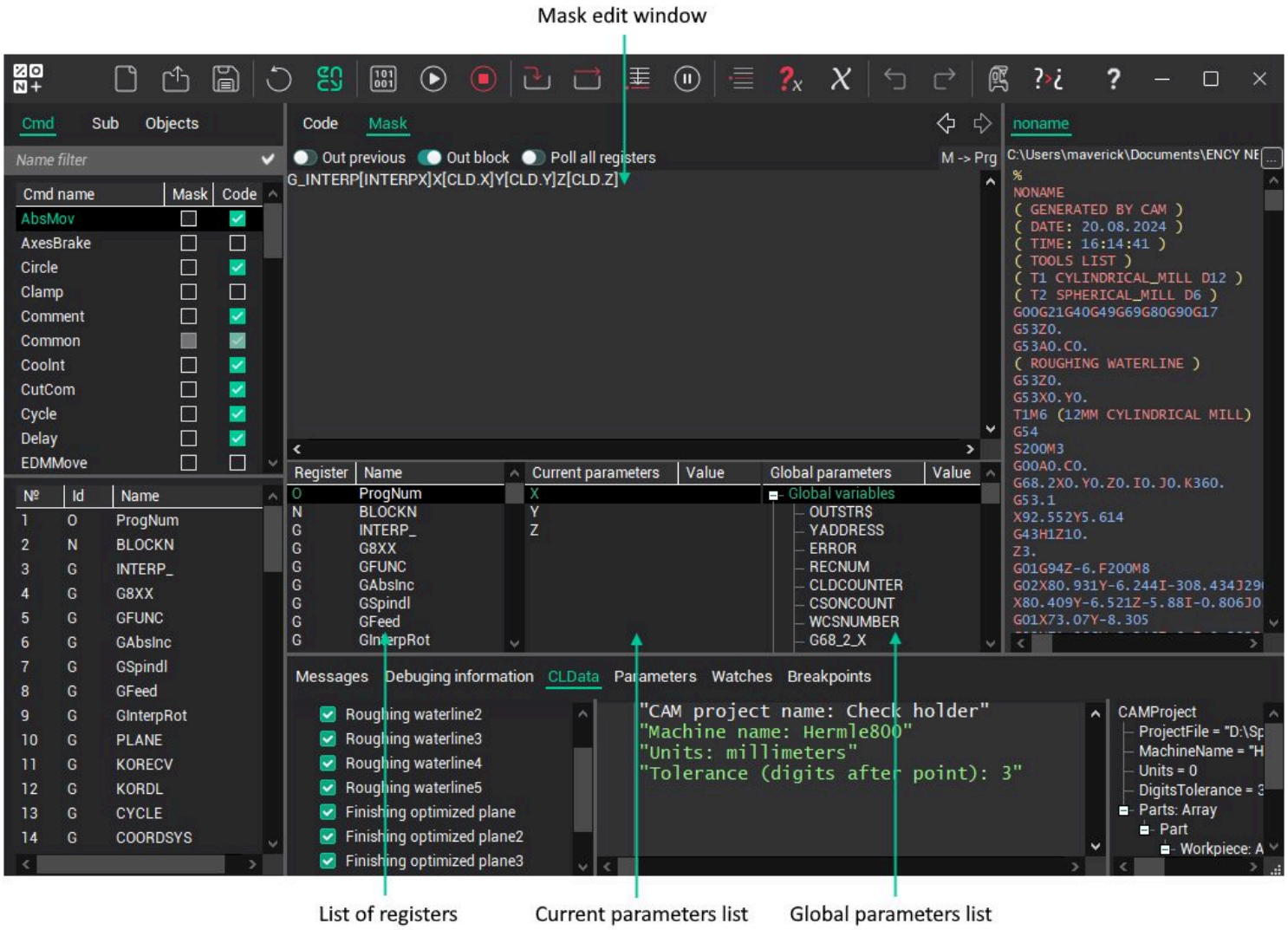
The corresponding statements in the command processing programs generate the block of the NC-program, moreover only the identifiers and values of the registers, which are changed since previous block, will be written in the current block.

The main window

The main window:



The Main toolbar are placed in the top part of the window. The list of command-processing programs and the list of registers are placed in the left part. There are the switching pages with editor of masks and editor of the command-processing program in the center. Except editor on bookmark, Mask is located lists of registers of local and global parameters. In the bottom – there are switching windows of the system messages, debug information, the list of the files, which contain the trajectory of the tool motion and the textual representation window of these files, the list of controllable variables and the list of break points.



See also

[The main toolbar]

[System settings]

[Editor settings]

[Defining the data about the NC-machine and CNC-system]

[Comments transliteration]

[The block structure and format definition]

[The masks for the machining commands translation]

[The programs for the CLData commands processing]

[Subprograms]

[The command processing programs compilation]

The work with the files of technological commands

The trial NC-programs generation

Programs debugging

The main toolbar

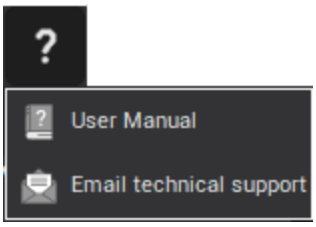


- – Creates the new postprocessor's tuning file. The system resets its state before creation.
- – Opens the postprocessor's tuning file, which is saved earlier. The system resets its state before loading.
- – Saves the postprocessor's tuning file with the current name.
- – Requests new CLData files from CAM system.
- – Opens the technological commands file, generated by CAM system.
- – Runs the compilation of the command processing programs.
- – Runs the generation of the NC-program from the files of the tool motion trajectory. If the command processing programs aren't compiled, the compilation will be activated first.
- – Break NC-program generation.
- – Generate NC-program with stopping in subprograms.
- – Generate NC-program without stopping in subprograms.
- – Generate NC-program up to current position in a CLData.
- – Pause NC-program generation.
- – Add breakpoint in a handler of technological commands or CLData files.
- – Adds a new variable to the list of watches.
- – Compute variable or expressions. The evaluation is possible only in a debug mode.
- – Cancel the last change made in the program of a handler technological command or a mask.
- – Return the last change made in the program of a handler of a technological command or a mask.
- – Opens the window for the data about NC-machine and CNC-system.
- – Open transliteration properties dialog. Application toolbar placed on the top left panel of the main window. There are drop-down menus associated with 2 buttons.



-  – Opens system setup window.
-  – Information about Postprocessors generator.

Help menu consist of actions that allow to get answers to emerging questions.



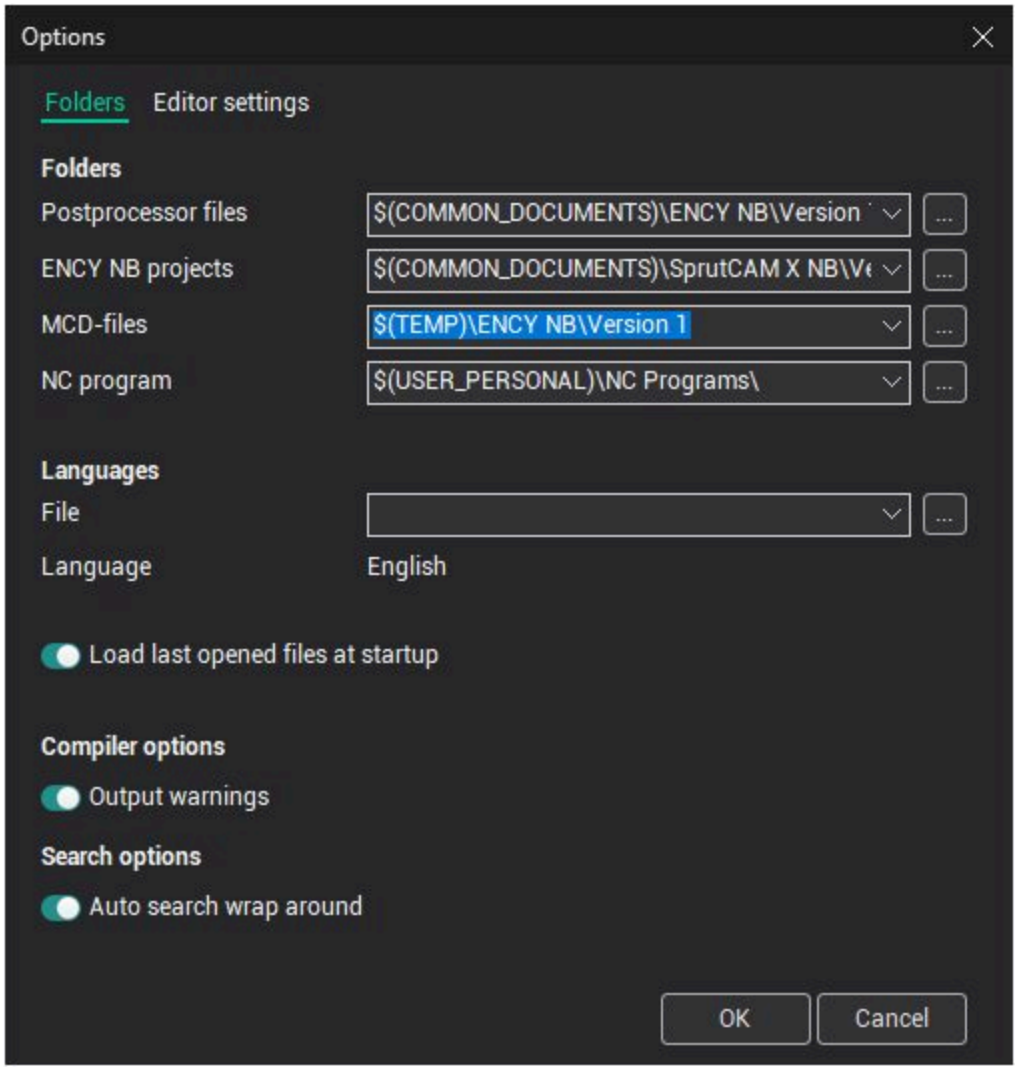
-  – Shows postprocessors generator help.
-  – Prepare a message to the technical support service of the CAM system vendor.

See also

[The main window]

System settings

The window of system settings can be activated from main menu. The default paths for the system files and languages are defined here.



The postprocessor adjustment files for various NC-system (*.sppx files) are loaded from the Postprocessor files directory. New postprocessor tuning files will be saved in this directory.

The CAM system projects are loaded as default from the CAM system project directory. The technological commands files (*.mcd), which are linked with opened project will be loaded from the corresponding paths, described in the project file.

The separate technological commands files (*.mcd) are loaded, as default, from the `MCD-files` directory.

Generated NC-programs are saved, as default, in the `NC-programs` directory.

To change program language it is necessary use Languages panel.

These paths may be edited manually or using dialogs, which can be activated by pressing ... button.

In a system, there is a preconceived variable, which one may be used for definition of the conforming folders:

At determining actual names of folders during operation the indicated variable will be substituted by the conforming full path at the moment of activation of a system or at the moment of the last editing the system settings.

For load last opened *.SPPX file at activation of the generator of postprocessors there is a check box Load last opened file at startup.

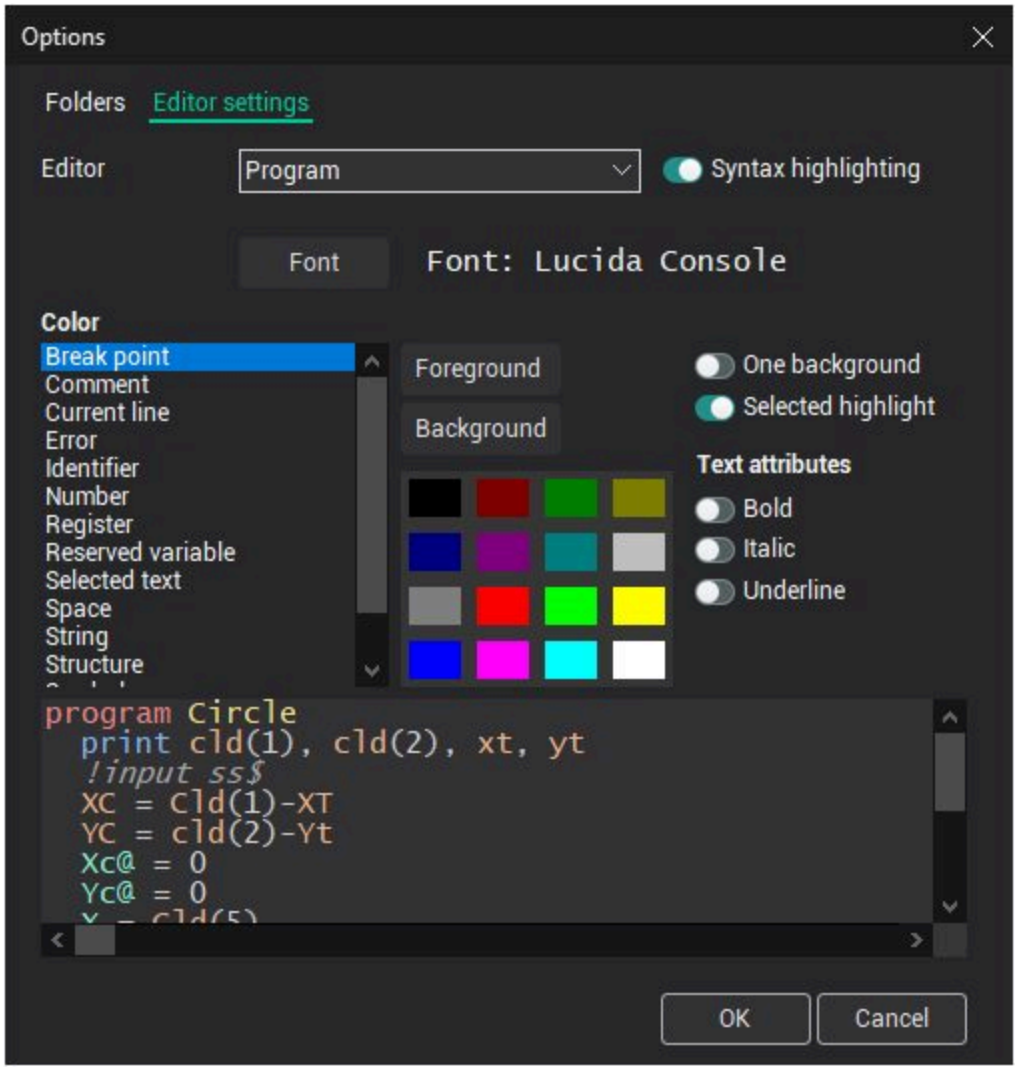
See also

The main window

Editor settings

Programs of technological commands consist of identifiers, the numbers, identifying reserved variables and functions, commenting etc. For each of these element blocks it is possible to adjust the color and type style, a background color.

An editor settings window can be activated from main menu. In a window, the font and illumination of syntax for the editor of programs of technological commands, windows of a text mapping of CLData files and windows of map of a NC-program adjusted.



For definition of customizations in a falling out list Editor, it is necessary to select one of editors: Program, CLData or the NC program. For the selected editor on the panel Font sets a name and a font size.

On the Color panel for each element block it is possible to assign color of the text, a background color of the text, type style.

For fast discoloration of a background of all element blocks, there is a check box One background. At selecting check box, the background color of all groups is substituted on a background color of a current

element.

Selected highlight feature will highlight selected word throughout the editor.

The OK button closes the window and saves all modifications. The Cancel button closes the window and discards all modifications.

See also

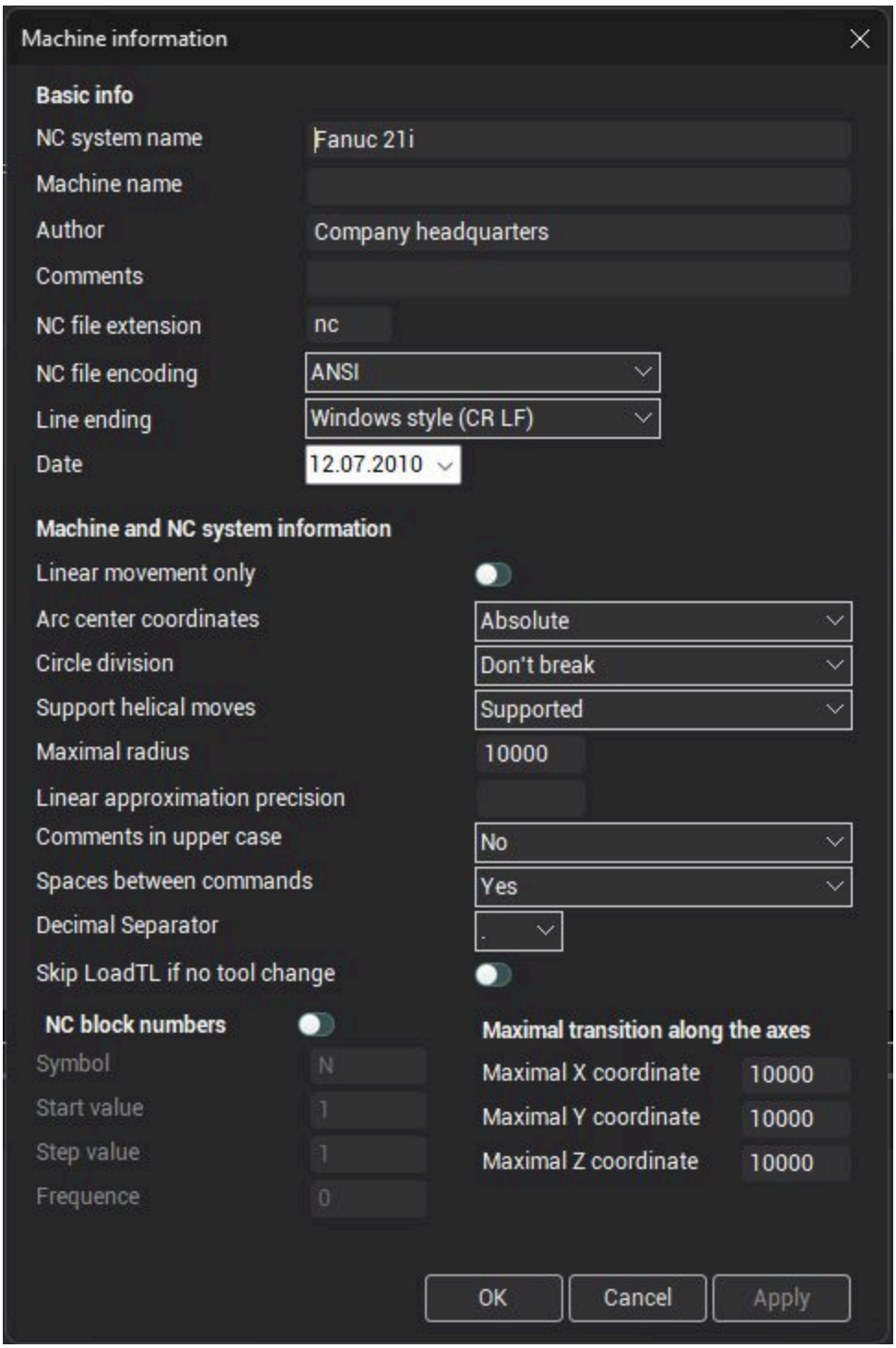
The main window

Machine parameters

No content in this page. See child topics.

Defining the data about the NC-machine and CNC-system

The editing window of the data about the NC-machine and the CNC-system can be opened by pressing the button on the main toolbar.



The name of NC-machine, the name of CNC-system and the extension of NC-program files are displayed in the Basic info panel. The name of NC-machine and the name of CNC-system are informative

parameters only. All NC-programs, which will be generated using edited tuning-file, will be saved with the specified extension.

On the Machine and NC system information panel defined corresponding parameters:

In a field Arc center coordinates, the mode of the definition of circle center is determined. If the relative mode of the definition of center of a circle in the variables XC, YC, ZC are set concerning for current selected point. If the absolute mode of the definition of center are set in absolute values is selected.

In a field Circle division is offered to select a mode of introducing of arcs in a NC program (a quarter, half or a full circle). The postprocessor during operation arrests intersection arcs of quadrants and, if necessary, dissects arcs into halves or quarters.

In a field Support helical moves are set with the information on support by a CNC system of helical moves. If helical moves are not supported, the postprocessor automatically approximates helical arcs on lines and shapes commands of linear movements.

In a field Maximal radius is entered a value of maximum radius of the arc bolstered by a CNC system, at overflow which one the postprocessor substitutes arcs linear movements. The value 0 in it a field disconnects monitoring, and the system will not approximate an arc at any value of radius.

For an event when the CNC system at all does not support arc interpolation and helical moves, check box Linear movements only is stipulated. In this case, all arcs without dependence from fields Support helical moves and Maximal radius are dissected into linear movements.

The capability stipulated to shape Comments in upper case and to place Spaces between commands of NC program.

For forming a block number there is a group of fields NC block numbers. If the corresponding checkbox on the panel is not set, then auto-numbering frames feature is disabled. If this checkbox to include, it will become available a number of fields to set up automatic numbering. The group consists of fields Symbol, Start value, Step value, Frequency. In a field Symbol is entered the identifier of the register or a symbol injected in block before a value of a block number. If this field is specified identifier register, then there is the possibility of additional control over the frame number by setting the format of the register (for example, if the frame number will go beyond the range of possible values of the register, then the numbering starts again). In the case of automatic numbering blocks are numbered since Start value. After each output of block to a block number Step value is added. The step value is available not only from the window Basic parameters, but also in the program through the predefined variable BlockStep.

Block number outputs with the given frequency. If frequency is equal to 1 the block number outputs in each command of forming of block if 2 – through one command, 3 – through two commands etc. If frequency is equal to zero the block number is not output.

Maximal transitions along the axes allow inspecting a correctness of a NC program. At overflow of as much as possible admissible transition on one of coordinates, the conforming warning will be show.

Decimal separator symbol which will be output into NC-block in place of decimal separator in numbers.

Skip LoadTL if no tool change - skips calling the LoadTL command handler if it does not actually change the tool relative to the previous operation. When option is enabled and if the "ToolChanged" property of the LOADTL command has value 0 (false), the LoadTL procedure for this command will not be called.

The OK button closes the window and saves all modifications. The Cancel button closes the window and discards all modifications.

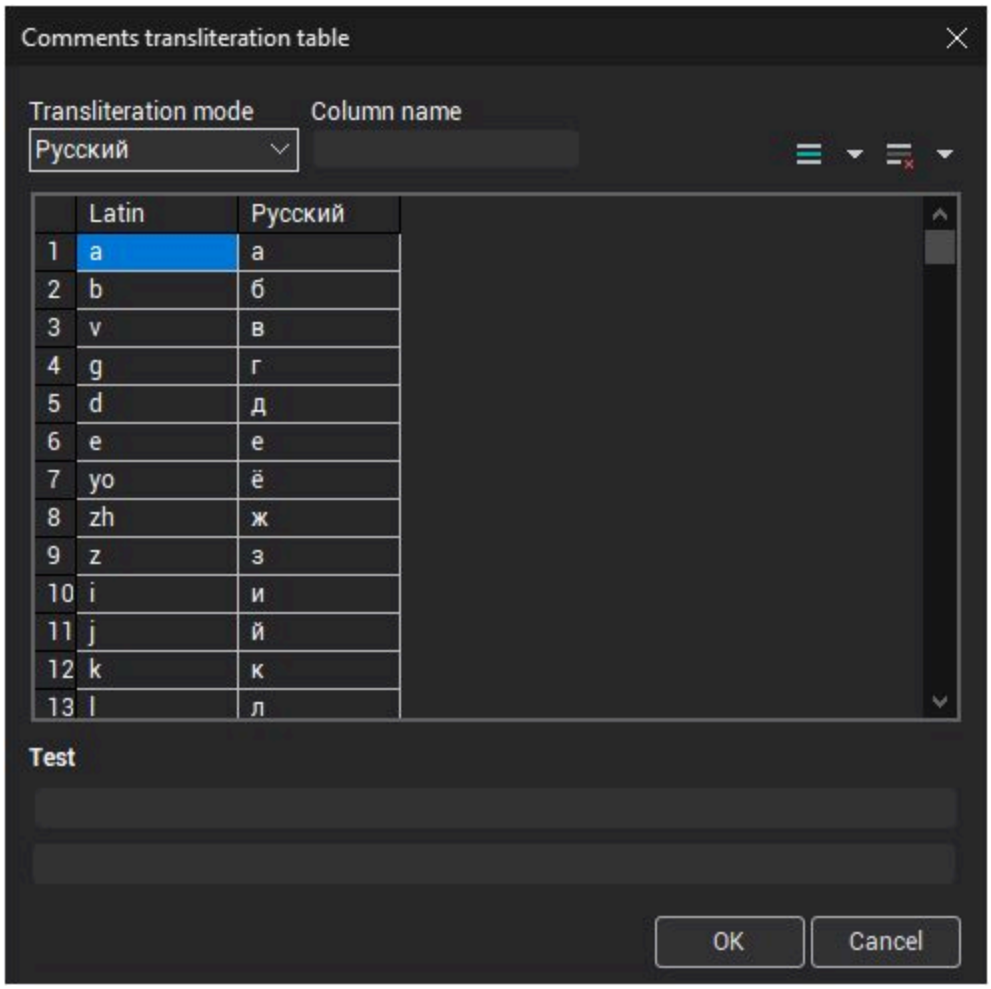
See also

The main window

Comments transliteration

Comments transliteration operation is used by postprocessor to transliterate comments in NC-program (passed through COMMENT technology command) that contain symbols not supported by the CNC system (for example non ASCII symbols) into literals that contain only the symbols that are acceptable for the CNC system.

Transliteration parameters are set by pressing the ?>¿ button on the main toolbar.



Transliteration mode set the original language of the comments (the active column in the transliteration table). Set to Off by default.

Each language that can be transliterated is represented by a column in the transliteration table. The first column Latin is the main column which symbols will be used to transliterate symbols from other languages. The rows of the table define which symbols in the active language will be transliterated (rows of the active column) and specify the transliterated symbols or symbol combinations (the Latin column).

The transliteration table can be modified at any time, you can add, remove or rename columns, and can add, remove or modify the rows and cells of any column. Use the Column name edit field to change the name of the column. Use the following buttons to add and remove rows and column of the table:

-  New row – adds new row after selected;
-  New column – adds new column after selected;
-  Delete row – deletes selected row;
-  Delete column – deletes selected column.

The Test panel is used to test the transliteration settings live. When you type some text into the upper edit field, the system writes the transliterated text into the lower field.

See also

The main window

The block structure and format definition (Register list forming)

To define the structure and the format of a block, it's necessary to form the list of the registers and to define their parameters.

Block of NC program consist of words. Each word contains address and value. Address – letter (sometimes several letters), Value – is number present in definite format. The determined Register is connected to each address in the postprocessor.

The concept of the Register of the postprocessor integrates following properties:

- the identifier of the register (the address in a NC program);
- the format of an output of a value of this address in block;
- current value conforming to the address;
- the previous value conforming to the address;
- a name of the register (post-processor variable associated with the address for accessing both the current and the previous value of the register).

The NC-program blocks are formed automatically when the OUTBLOCK and FORMBLOCK statements are performed. The NC-program block is formed by the system according to the following algorithm. The system examines the registers sequentially; if the current register value differs from its previous value, then the register will be written in the block and its current value will be assigned to its previous value, else the register will not be written in the block.

When the system writes the register in the NC-program block, it writes the register identifier first, and then it writes the register value. The value will be written in the block according to the format, described for this register (the length and the precision of the register value, the presence of decimal point, of sign and of leading and insignificant zeroes).

Thereby, to define the structure and the format of the block, it is necessary to fill the registers list and define the registers properties. The registers must be placed in the list in this order, in which they and their values must appear in the blocks of NC-program. This rule is correct for the processing programs. If the block is formed with, the mask using then the sequence corresponds to the mask.

Note: Different registers must not have same names, but they may have the same identifiers (the symbols, which will be written in the block before the value of register). This allows creating the separate register for each group of the functions with some type.

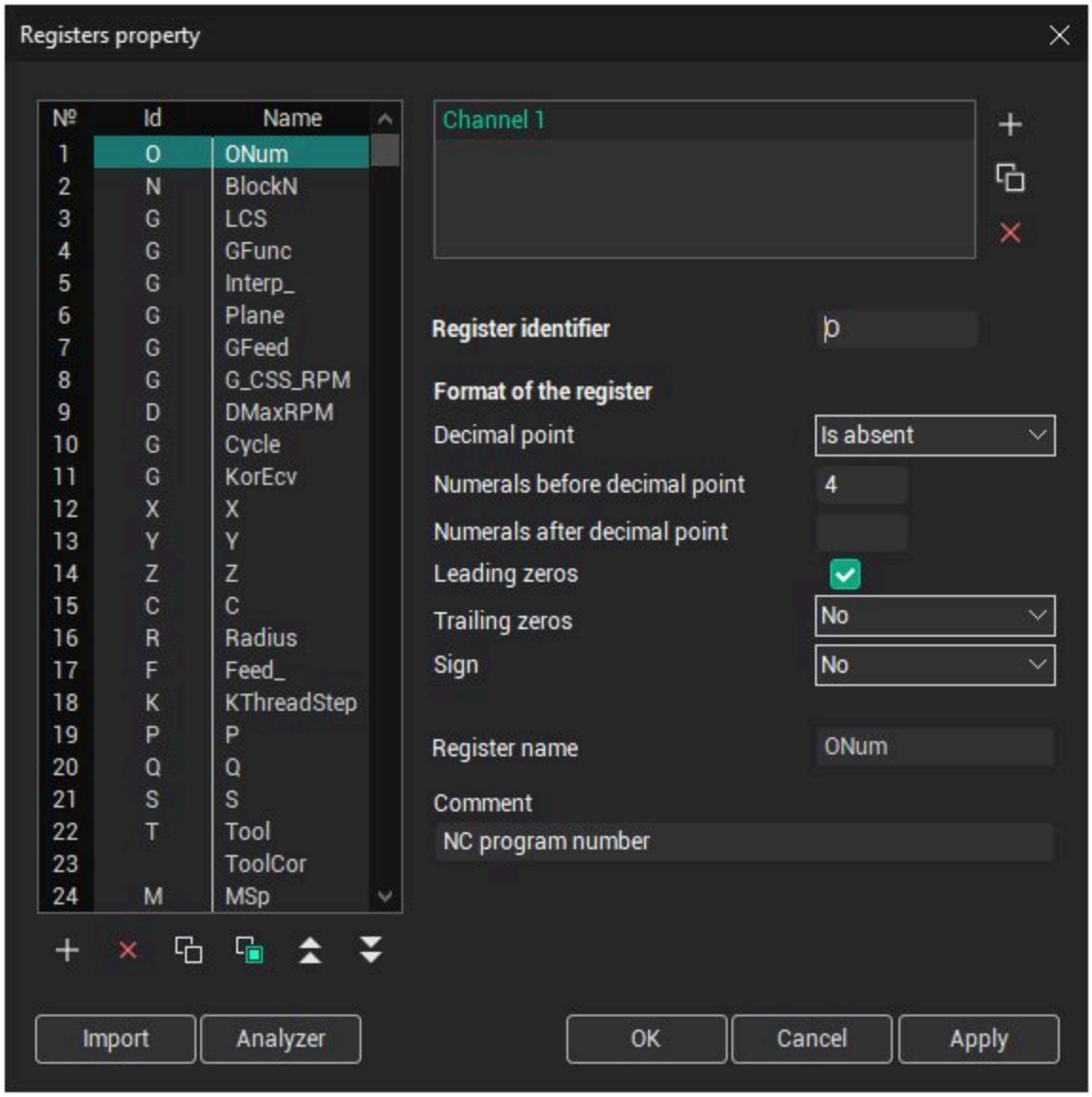
For example, we create the register with the name ABSOLUTE and with the G identifier for the preparative function of the switching between the absolute/incremental coordinate systems. Suppose also, we create the register with the name INTERP and the same identifier G for preparative function of

positioning, linear interpolation, circular interpolation with direction. This allows us: firstly, to write both these commands in the same block of NC-program, if it is necessary, (for example, `N190 G91 G1 X50 Y30`). Secondly, to trace the current transitions mode (positioning, linear or circular interpolation with direction), to determine the current coordinate system (absolute or incremental) by examining the current values of registers ABSOLUTE and INTERP.

The list of registers is shown in the left part of main window.

№	Id	Name
1	O	ProgNum
2	N	BLOCKN
3	G	INTERP_
4	G	G8XX
5	G	GFUNC
6	G	GAbsInc
7	G	GSpindl
8	G	GFeed
9	G	GInterpRot
10	G	PLANE
11	G	KORECV
12	G	KORDL
13	G	CYCLE
14	G	COORDSYS

The register properties edit window is opened when the register is added, when the item Properties in the context popup menu is selected or by double-clicking by left mouse button on the register in the list.

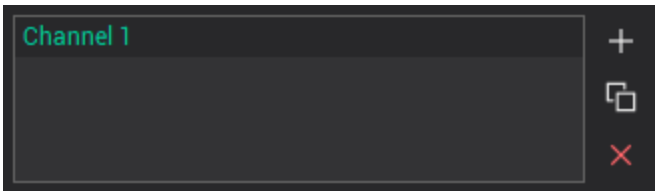


Register list is shown at the left part of window. Edit list buttons place below:

-  – inserts the new register after selected one, the window to edit the properties of a new register will be opened automatically;
-  – deletes selected register;
-  – copies the selected register into the clipboard;
-  – inserts the register from the clipboard after the selected one. The window to edit the properties of inserted register will be opened automatically;
-  – move the current register up on a list;
-  – move the current register down on a list.

To change the position of the register in the list, it is necessary to press the left mouse button on the register and drag it in the desired position, holding the left mouse button down.

If you have a multi-channel machine, you can control the registers of a specific channel by selecting it in the channel list.



You can also manually add channels , copy from an existing channel to a new one , or delete the current channel .

Following fields can be defined in this window:

- Register identifier – the symbols, which will be written in the NC-program block before the register value.
- Decimal point – this field can have following values:
  - Is absent;
  - Is present – if this item is chosen, then the decimal point will be present, if the register value has fractional part;
  - Is present anyway.
- Numerals before decimal point – maximum quantity of signs in the whole part of number.
- Numerals after decimal point – maximum quantity of signs in a fractional part of number.
- Leading zeroes and Non-significant zeroes – defines the zeroes output mode before and after the register value.
- Sign – defines the output mode for the sign of the register value. Following options are available:
  - No;
  - "-" only;
  - "+" and "-" always;
- Register name – the name of the register, which is used by the command processing programs.
- Comment – comments to register.

The Import button is intended for import of a list of registers from postprocessors of SPPX, SPP and PPP (the format of the old version), and as from SurfCam postprocessors.

For forming a list of registers based on a NC program, the Analyzer button is intended. Thus the text of the indicated NC program and all retrieved addresses is analyzed are added in a list of registers.

The OK button closes the window and saves all modifications. The Cancel button closes the window and discards all modifications. The Apply button save all changes, window of registers is not closed.

The current value of any register is available from program code by register's name as a usual variable. To get previous value of the register you need to write the name of register with @ char. For example:

```
program AbsMov
  Len1: Real
  Len1 = sqr((X-X@)^2 + (Y-Y@)^2 + (Z-Z@)^2) ! Here X, Y and Z are current values of
registers, X@, Y@ and Z@ are previous values of registers
end
```

Also you can get access to the registers from program code with RGS operator in form of Rgs[Index in list] or Rgs["RegisterName"]. For example for exact list of registers that shown on the picture above:

```
program AbsMov
  Len1: Real
  Len1 = sqr((Rgs[3]-Rgs[3]@)^2 + (Rgs[4]-Rgs[4]@)^2 + (Rgs[5]-Rgs[5]@)^2) ! Here
Rgs[3] is the current value of the register X, Rgs[3]@ is the previous value of
register X
end
```

The full syntax that can be used with RGS operator listed below. Inside square brackets you can use index of the register in list (numerical constant or variable) or name of the register (string literal constant or variable).

- `RGS.ItemCount` - returns count of the registers in the list.
- `Rgs[<Index or Name>].STR` - string ID of the register which is used from G code side.
- `Rgs[<Index or Name>].NAME` - string name of the register which is used from postprocessor's code side.
- `Rgs[<Index or Name>]` or equivalent `Rgs[<Index or Name>].DATA` - current value of the register.
- `Rgs[<Index or Name>]@` or equivalent `Rgs[<Index or Name>].OLDDATA` - previous value of the register.
- `Rgs[<Index or Name>].NUM` - returns the index of the register in the list of registers.

Example that shows advantages of using RGS instead of name of the register is below.

```
program AbsMov
  ToolAxisIndex: Integer
  case PlaneGNumber of ! Remember tool axis register index depend on current plane
    17: ToolAxisIndex = Rgs["Z"].Num
    18: ToolAxisIndex = Rgs["Y"].Num
    19: ToolAxisIndex = Rgs["X"].Num
  end
  X = CLD.X ! Fill X, Y and Z registers' current value from CLDATA
```

```
  Y = CLD.Y
  Z = CLD.Z
  if Interp=0 then begin ! If we have rapid G0 movement case only let's we want make
safe moving down
    if Rgs[ToolAxisIndex] < Rgs[ToolAxisIndex]@ then begin ! If the tool is
moving down
      Rgs[ToolAxisIndex]@ = Rgs[ToolAxisIndex] ! Then we prevent output of tool
axis movement
      OutBlock ! So here we output movements in active plane only
      Rgs[ToolAxisIndex]@ = 999999 ! Now we must assign any unmatched number to the
old value of the register to
    end ! provide guaranteed output of movement along tool axis on the next OutBlock
  end
  OutBlock ! Output movement to the G code
end
```

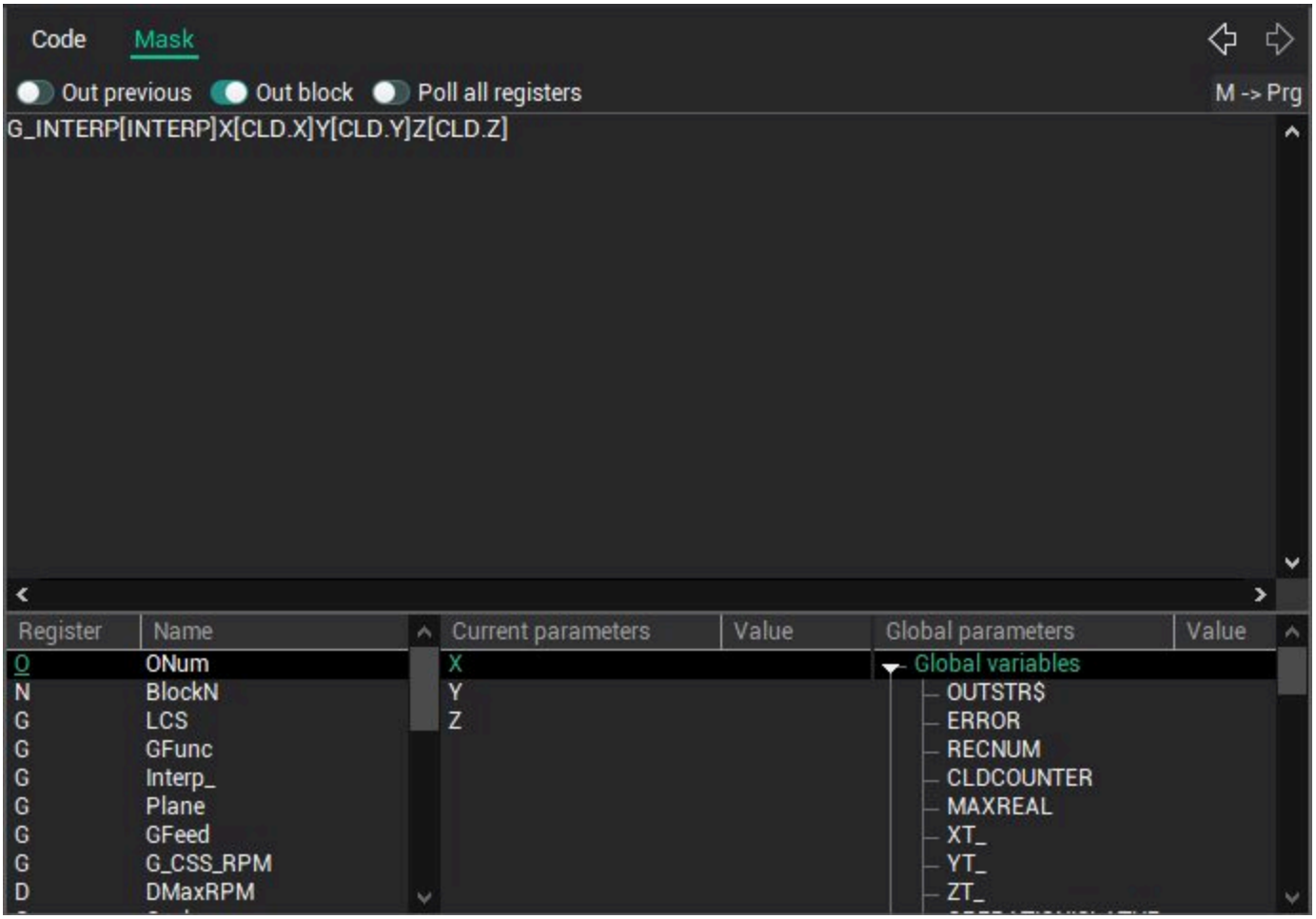
See also

[The main window]

The masks for the machining commands translation

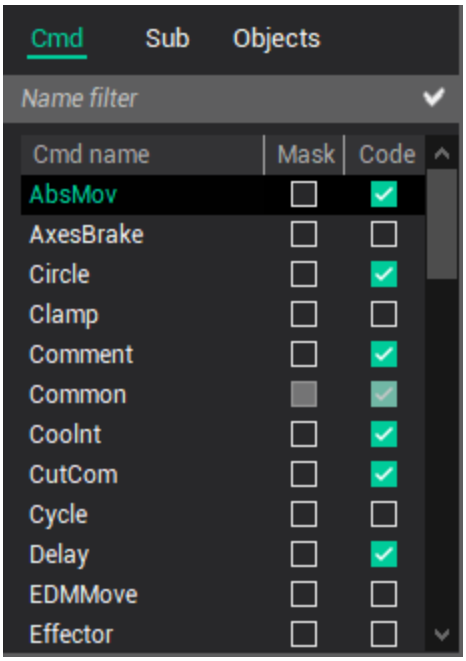
The usage of the masks allows defining the transformation method from the CLData machining commands to the NC code line quickly and simply. Any CLData command has the corresponding mask, in which the words and the values for the words are defined.

The mask editor is located on the Masks page.



For the simplifying the process of the mask definition there are the registers list, current command parameters list and the global variables list in the bottom part of window. The double click on any element of these lists inserts the element to the mask text. Above the mask text field is located a checkbox panel that controls the mask processing.

The mask activating is performed by the tick setting on the CLData commands list.



The masks definition rules is described in details in the Postprocessor masks chapter.

CLData commands that are not handled by this postprocessor can be hidden by selecting the appropriate item from the popup menu.

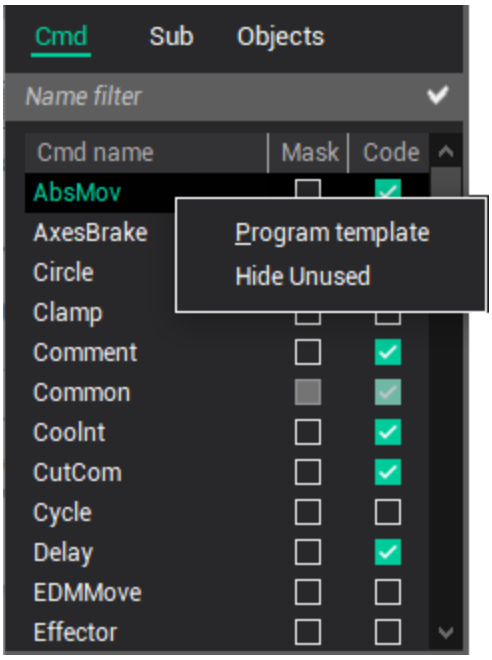
See also

The main window

The programs for the CLData commands processing

Like the masks, the programs serve to define the transformation process from the CLData command to the NC code line. The programs can expand the mask or can be used instead the masks. Both the mask and the programs have merits and demerits. The programs is the flexible and powerful tool for the realization of the very complicated transformation. However, the learning of this tool requires the programming experience.

Any CLData command has the corresponding processing program.



These programs are designed using the special problem-oriented language and can contain the mathematical expressions and the functions, statements for input/output, conditional statements, cycles, jump statement, subroutines calls, the statements to form the NC-program blocks and the statements to work with the technological commands file. This language is described in detail in the Language description chapter.

Each program begins from the header, which consists of the PROGRAM keyword and the program name, and terminates by the END keyword. The program name is coincident with the command name, processed by it. When the program is called, the parameters are passed to it via the predefined CLD array and special Cmd operator.

The program activating is performed by the tick setting on the CLData commands list in the code column. If the program is not active then it will not be translated and executed even if the text exists. The program COMMON is exclusion. It is translated and executed always and at the first. This program is intended to define the global variables i.e. variables that are accessible from any subprogram.

The empty program is generated by double click in the list of technological commands.

When the right mouse button is pressed in the programs list, the context popup window will appear. This popup contains following items:

- Program template - the program template is initialized as blank, including a header and an end operator;
- Hide unused - unactivated programs will be hidden from view.

Programs have the capability to use subroutines, which are organized in a separate tab section. When the right mouse button is pressed in the subprograms list, the context popup window will appear. This popup contains following items:

- Insert – adds a new technological command processing program. The window to input the program name will be opened.
- Delete – deletes selected program.
- Rename – renames current program. The window to input the program name will be opened.

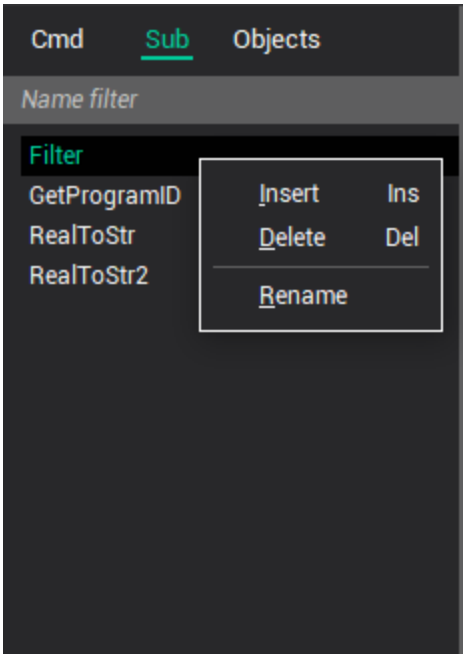
The editor of the selected program is placed in the center of main window. When other program is selected in the list, its text will be displayed immediately in the edit window.

See also

The main window

Subprograms

Subprograms tab holds the list of user defined subprograms and the Filter subprogram. Use the context menu to add, remove or rename subprograms. The code of the subprogram can be viewed or edited in the code editor. The code editor displays the code of the selected subprogram.



The name of the subprogram can be any legal string identifier, provided it is not the name of a technology program or already used by other subprogram.

The principles of definition and syntax of subprograms are described in the Language description, Subprograms topic.

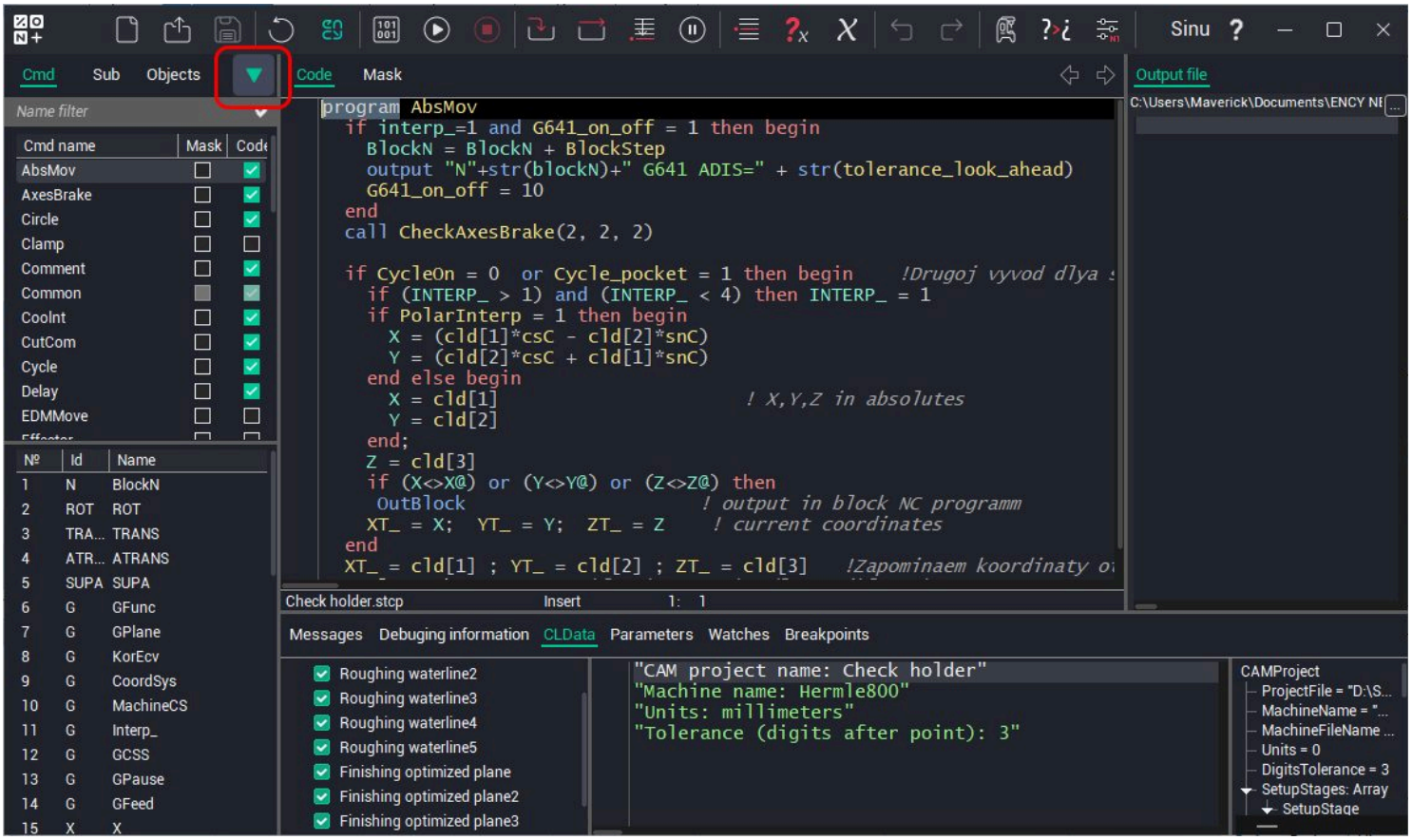
See also

The main window

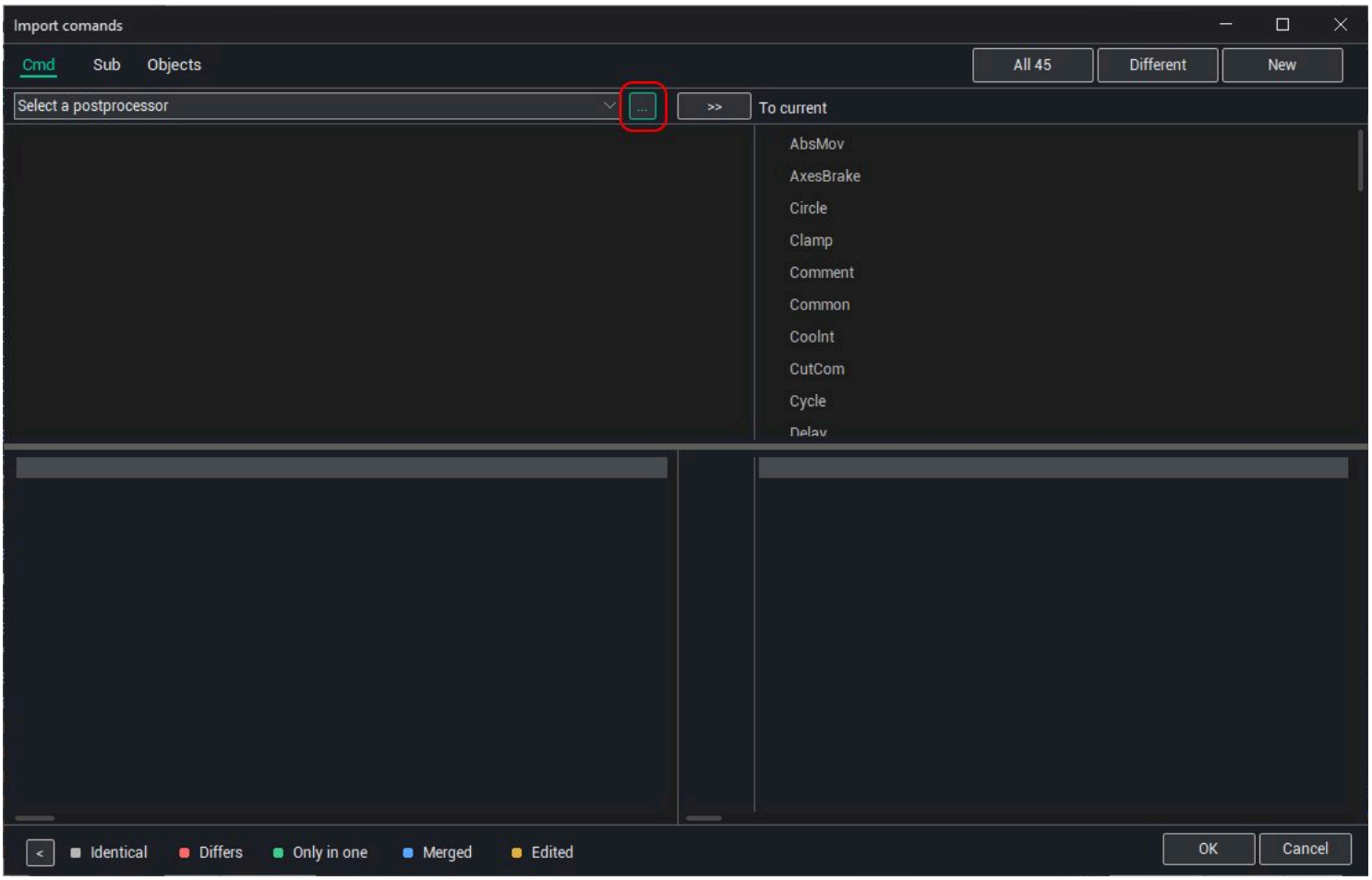
Importing commands, subprograms, and objects

The Import window lets you compare the current postprocessor with another postprocessor and import commands, subprograms, and objects from it. You can import a complete item or only selected differing code fragments.

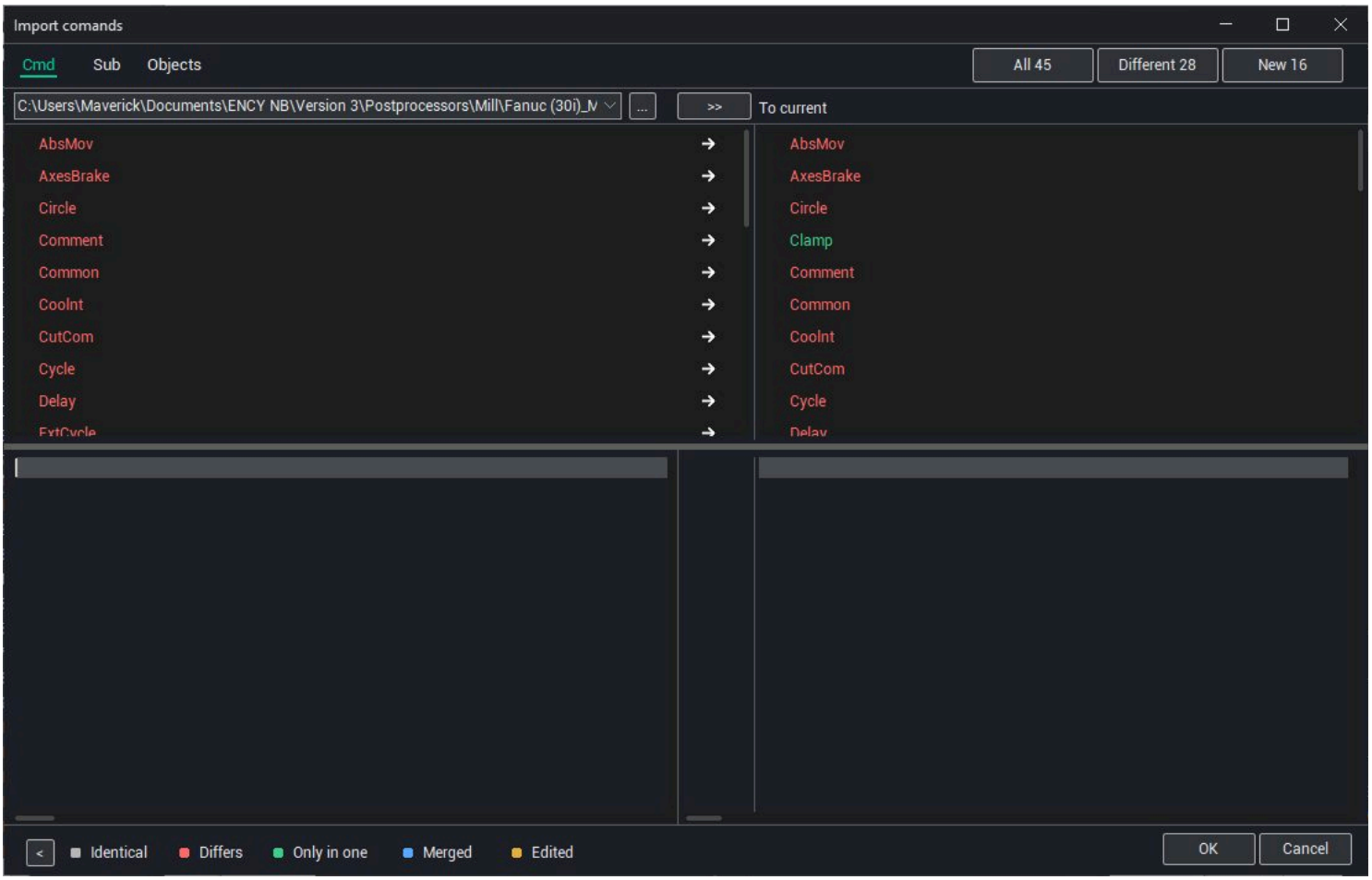
Open the Import window by clicking the arrow button in the Cmd, Sub, or Objects selector.



Click the postprocessor selection button and select the postprocessor from which you want to import items.



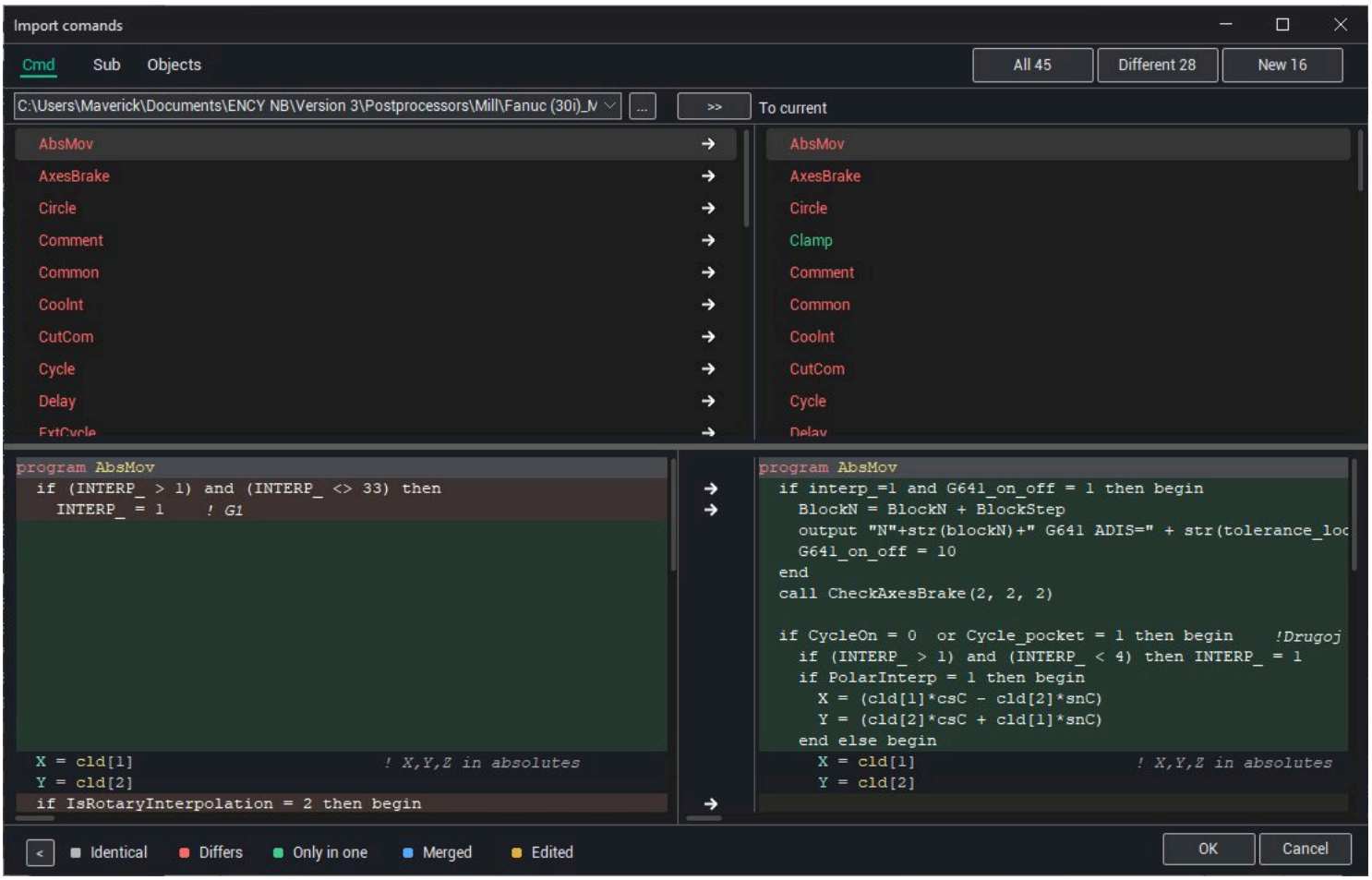
After the postprocessor is loaded, the left side of the window displays its items, and the right side displays the items of the current postprocessor.



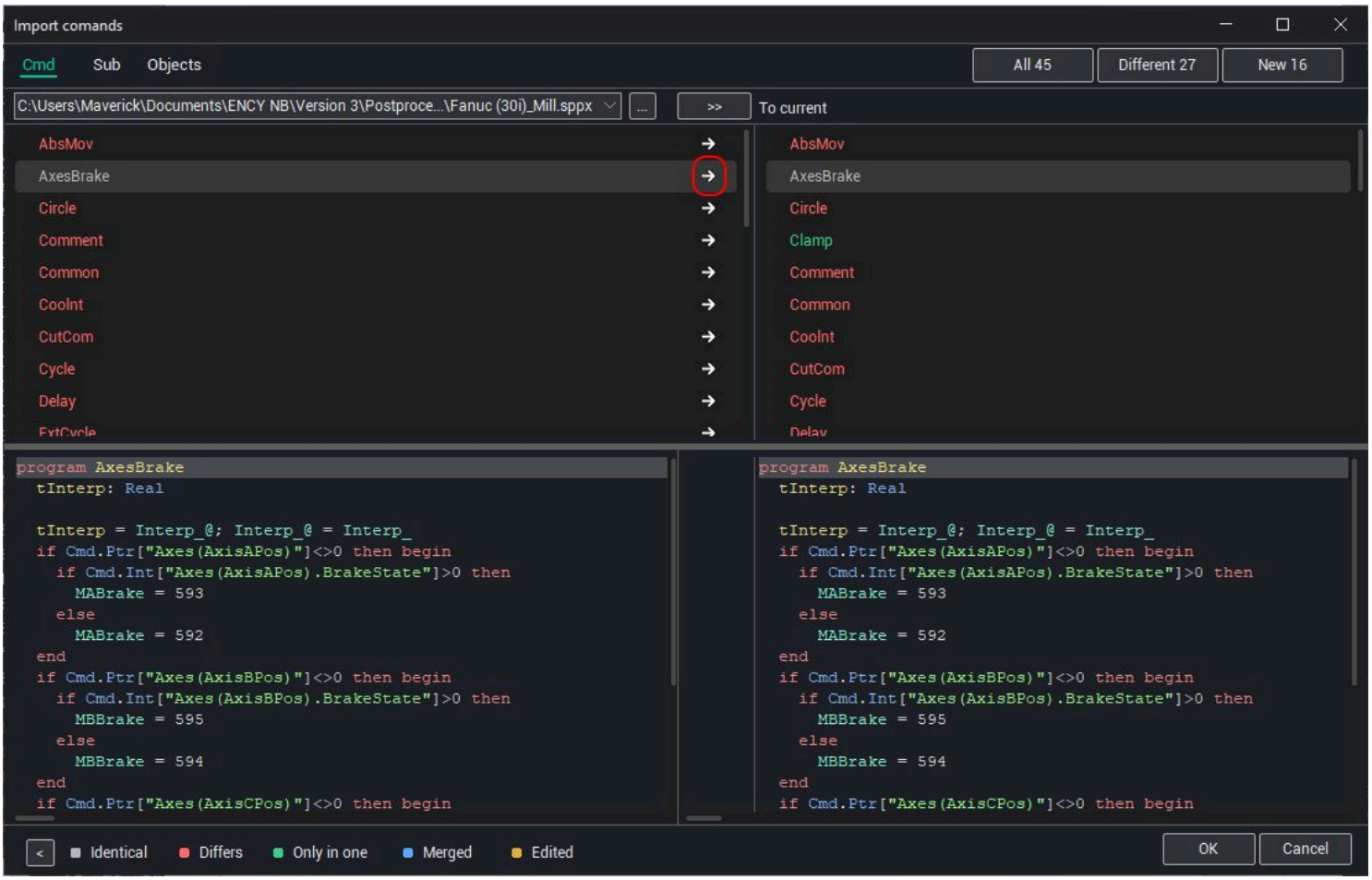
The item color shows the comparison result:

- red -- the item exists in both postprocessors, but its contents differ;
- green -- the items are identical;
- black -- the item is missing from the other postprocessor.

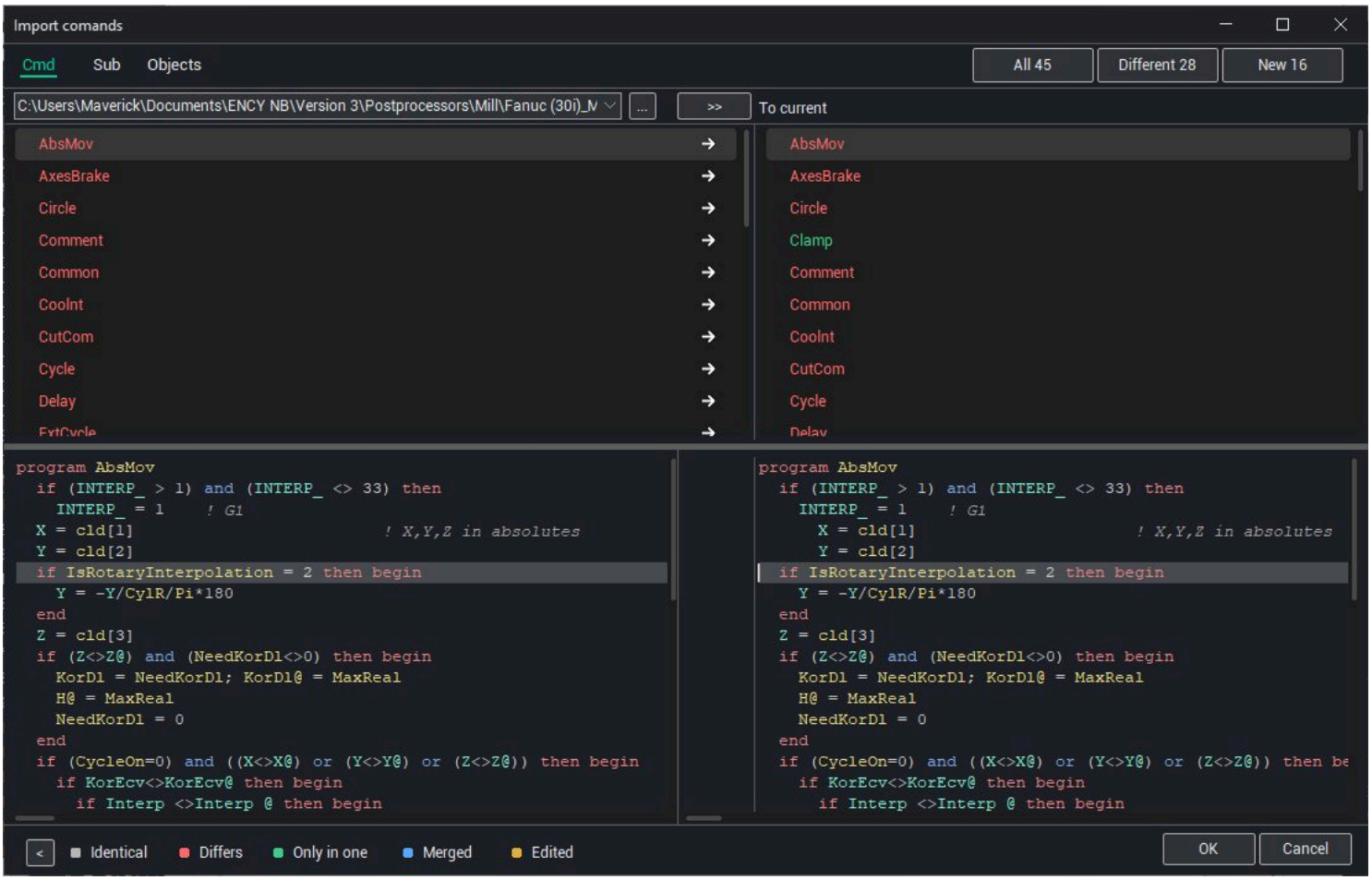
Select an item highlighted in red to view its code in both postprocessors.



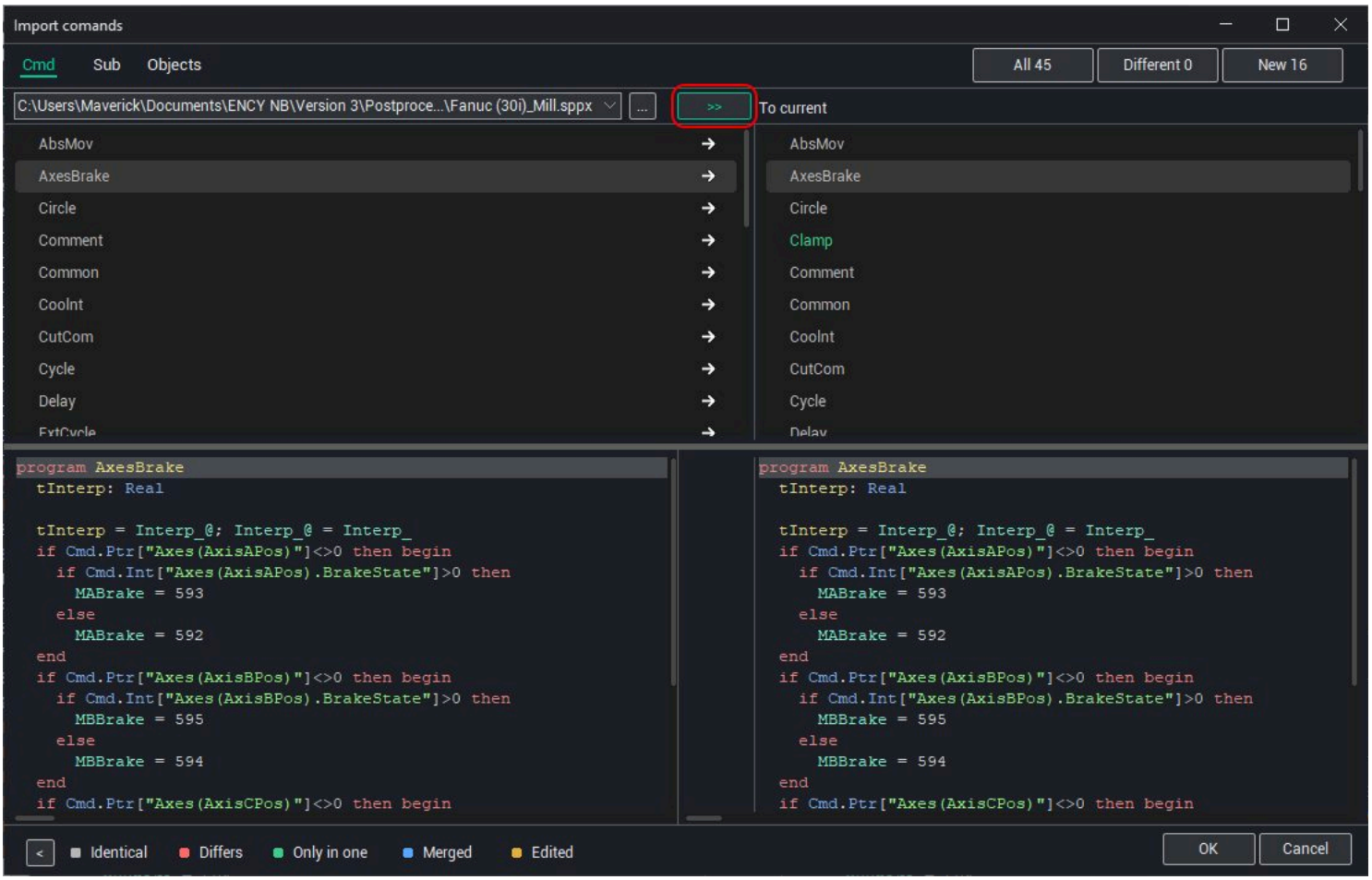
To replace an item in the current postprocessor completely with the version from the imported postprocessor, click the arrow next to the item name.



To import only one differing fragment, click the arrow next to that fragment in the code pane of the current postprocessor.



To replace all items in the current postprocessor with the versions from the imported postprocessor, click Merge all to current.



You can load another postprocessor for comparison and then return to a previously loaded postprocessor. Select the required postprocessor in the drop-down list next to the load button.

See also

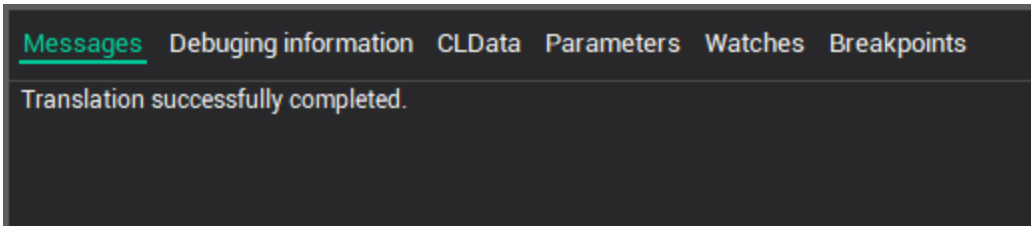
[The main window]

The command processing programs compilation

The compilation can be running by pressing the  button on the main toolbar or by pressing the [Ctrl-F9] combination.

Note: The activated program is compiled only.

The system messages about the compilation process are outputted in the messages page in the bottom part of the main window.



Errors occurring as a result of program translation are added to the system message list with the prefix [Error]. Double-clicking the left mouse button on such a message will load the original program text and set the cursor to the error position.

See also

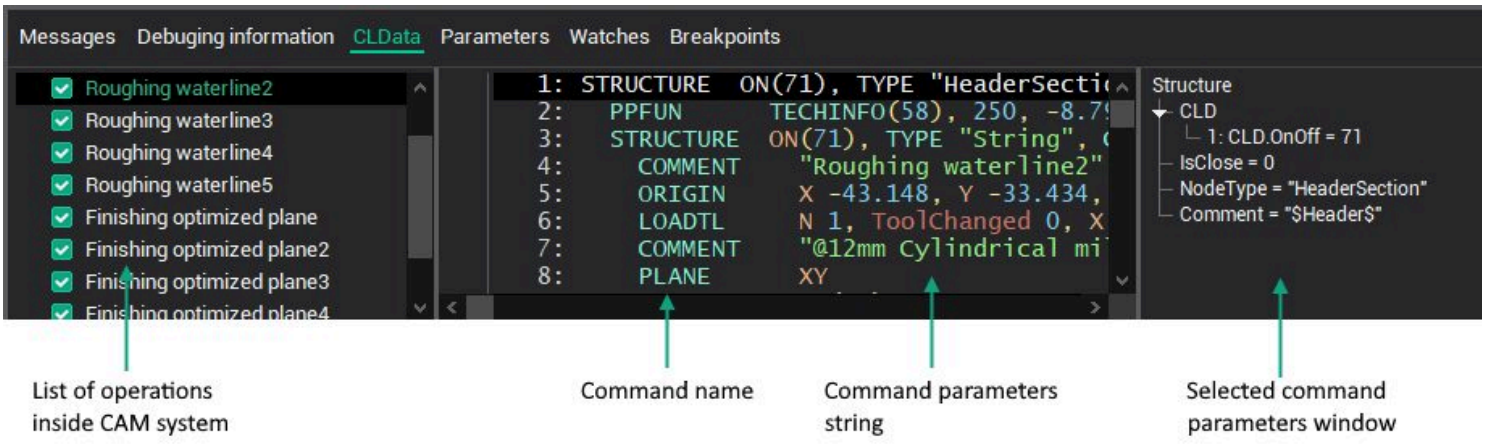
The main window

The work with the files of technological commands

The files of technological commands can be loaded from the CAM system project (*.stcp, *.stcx). The pressing the button or the choosing the corresponding item in the main or context menu loads the file.

When the files of the technological commands are loaded from the CAM system project, the project name, which contains these files, will be displayed on the main panel.

In the bottom part of main window on the CLData page the list of machining commands files is located. Any line of the list corresponds to a CAM system operation. Only ticked operations will form the NC code.



The operations list can be edited by popup menu. Click right mouse button in the list field to open popup menu. Menu contains the next items:

- Insert – adds the file of the technological commands;
- Delete – deletes selected file;
- Delete all – deletes all files.
- Update CLData – requests new CLData files from project opened in CAM system.

In the center of the CLData tab is located a text representation of technological commands (CLData commands). It displays a list of commands for a file that is highlighted in the list on the left. Each technological command in the CLDATA text is a separate line, consisting of the command name and its parameters. When the left mouse button is double-clicked on the command name, the mask corresponding to the command will be loaded to the masks editor and the program corresponding to the command will be loaded to the programs editor.

To the right of the list of commands technological command parameters window located. It contains a list of parameter names and their values for the selected command. To quickly obtain the value of a parameter in the program code, you must activate the appropriate parameter and select the Add parameter to code item in the popup menu (opens right mouse button click) or to double-click the left mouse button. The parameters are accessible via the CLD array or Cmd operator in the programs and via the current command parameters list in the masks.

See also

[The main window]

The test NC-code generation

The trial NC-program generation can be performed by pressing the  button on the main toolbar or by pressing the [F9] key. If the technological commands processing programs are not compiled at this moment, the compilation will be activated automatically. If there are errors during the compilation, the trial running will not be performed.

The loaded files of technological commands are the input data for the NC-program generation. The NC-program is formed using the technological files, which are marked by the flags in the list.

The algorithm of generation is described below:

1. The system reads the next command from the file of the technological commands.
2. The system analyses the command code and according to the command parameters define the predefined CLD array and predefined variables.
3. If the corresponding program for the command is activated then the program is executed.
4. If the corresponding mask for the command is activated then the NC code line is generated by mask.

Note: So for any command both program and mask can be activated. In this case the program will be executed first and mask later. If both mask and program are not activated then the command will be ignored.

5. If in result of the previous actions the NC-code line is generated then the subprogram Filter is executed. Usually this subprogram is used for the replacement of some elements in the NC block before the final output to the NC program.

The corresponding operators form the NC block from the mask or from the processing program. If the processing program is used then only the identifiers and values the same registers are output that was changed after the previous block output.

The NC-program will be saved in the NC-program text window (it is in the right side of the main window) and in the file with the name, defined in the field of the NC-program name. If this field is empty then program is not saved in a file. If the NC-program contains subroutines, then they can also be displayed in separate files, in this case will have additional tabs, switching to which you can view the generated subroutines text.

The debug information, which is formed in the processing programs, is written in the debug window.

If there are errors during the generation process, they will be written in the system messages window.

After the faultless NC code generation there is the link between the NC code and the CLData commands. By double click on the command name in the window of CLData commands the corresponding line in the

NC code will be selected. Conversely, by double click on the code line in the NC program window the corresponding CLData command will be selected.

See also

The main window

Programs debugging

Integrated debugger allows controlling the program execution. The program can be executed step by step with the checking the program code and the execution result. While debugging it is possible to enter into the subprograms or execute the subprogram in one step, to control the variable values and to view the debug messages. The debugger cardinally reduces the postprocessors designing time and lightens the bugs search and elimination.

Debugger functions:

- Program run – [F9];
- Run program to cursor – [F4];
- Step over (execute the current statement without the entering into the subprogram) – [F8];
- Breakpoint setting/resetting – [Ctrl+F8];
- Step into (execute the current statement with the entering into the subprogram) – [F7];
- Break the debugging – [Ctrl+F2];
- Add a new variable to the watch list – [Ctrl+F7];
- Evaluate expression – [Ctrl+F4].

Before the program is started the all modified subprograms are compiled. If there are no errors found while compilation then the program is started.

To start the program execution it is possible by any of commands: [F4], [F7], [F8] or [F9].

[F4] runs the program to the statement on the cursor position. If current string has not the statement then the program is executed to the end. The string where the execution was break is the debugger cursor that is selected by color.

If debugging is started then [F7] and [F8] executes the statement of a program where the debugger cursor is located else the debugger cursor is set on the first statement of the first program. Command [F7] as distinct from [F8] allows entering into the subprogram if the current statement is a CALL.

[Ctrl+F8] allows to set/reset a breakpoint. When the program execution arrives at the breakpoint then the execution will be paused with the debugger environment activation.

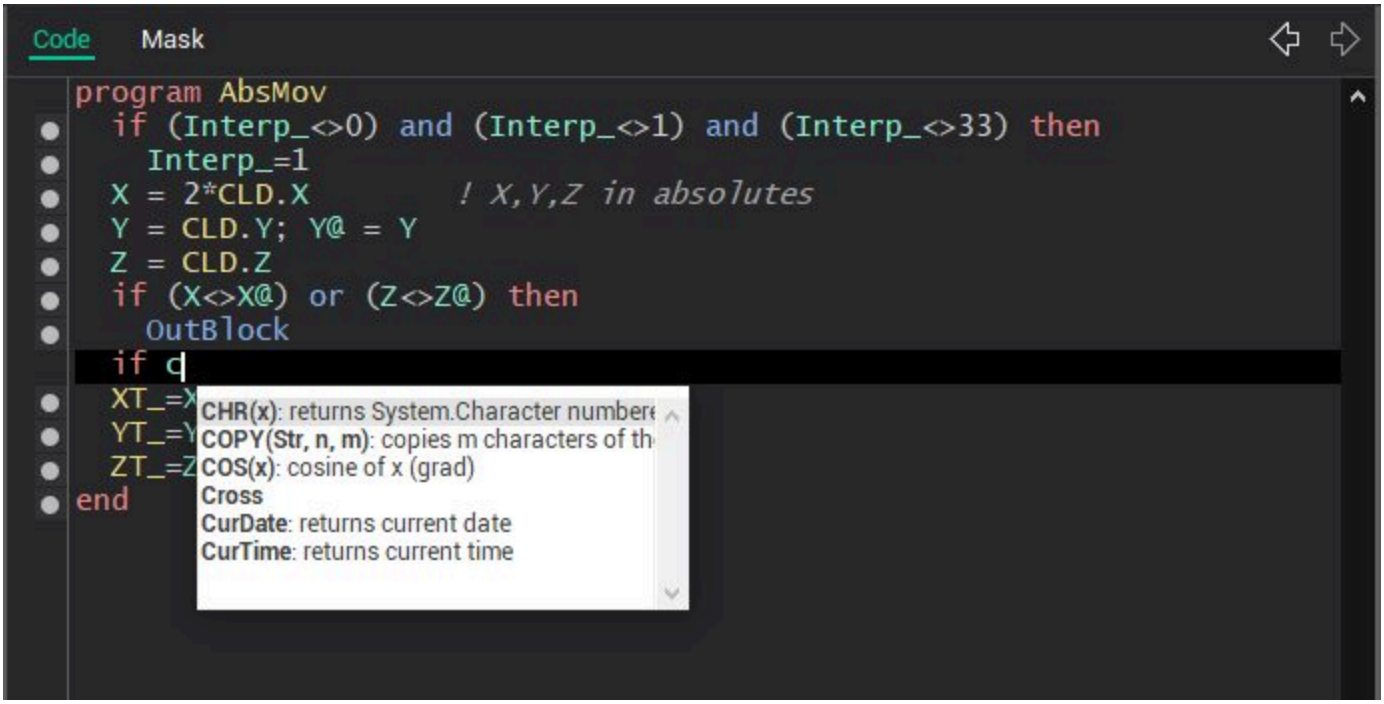
In any point of debugging process, it is possible to break the execution by [Ctrl+F2] or to continue the execution without debugging by [F9].

The changing of the lines number in the debugged program brings the incorrect indication of the current debugger cursor.

[Ctrl+F7] allows adding the variable or the expression to the watches list. If [Ctrl+F7] is pressed in the editing mode then variable name is taken from the current position of cursor. In the debugging mode then the mouse pointer is located under the variable then the variable value is highlighted in the hint.

To search for a word, double-click on this word and a new tab will appear with a list of all mentions of this word in all commands.

When editing code of programs to enter the names of predefined functions and variables is convenient to use a popup-hint. You can call it at any time while editing by pressing the key combination [Ctrl+space]. The window that appears contains a list of standard features, matching the already entered in the editor part of the word, and also shows a short description of these functions. After double-clicking on one of the rows in the list the function selected in it will be added to the current position of editor.



See also

[The main window]

Postprocessor parameters inquiry while the first using

Sometimes a dealer does not know some specific information about the user machine. It is possible to make the data inquiry. In this case the user will be asked about the NC-system information and inputted data will be saved in the postprocessor file.

An *.sppx file of a postprocessor it's a regular XML file. So you can edit it by a simple text editor (Windows' Notepad for example, or it is better Notepad++ to get syntax highlighting) to organize the data inquiry. There is a CommonDefinitions section (or node in XML terminology). It is shown below. It is a textual representation of an information you define in the Machine information window.

```
<CommonDefinitions>
    <LinearApproxPrecision IsSet="False">0.01</LinearApproxPrecision>
    <Circles>
        <Support>True</Support>
        <HelixSupport>True</HelixSupport>
        <Center>Absolute</Center>
        <Division>DoNotBreak</Division>
        <MaxRadius>10000</MaxRadius>
    </Circles>
    <XMax>100000</XMax>
    <YMax>100000</YMax>
    <ZMax>100000</ZMax>
    <UpperCaseComments>True</UpperCaseComments>
    <Spaces>False</Spaces>
    <DecimalSeparator>Point</DecimalSeparator>
    <BlocksNumbering>
        <Address>N</Address>
        <StartValue>100</StartValue>
        <StepValue>2</StepValue>
        <Frequence>0</Frequence>
    </BlocksNumbering>
</CommonDefinitions>
```

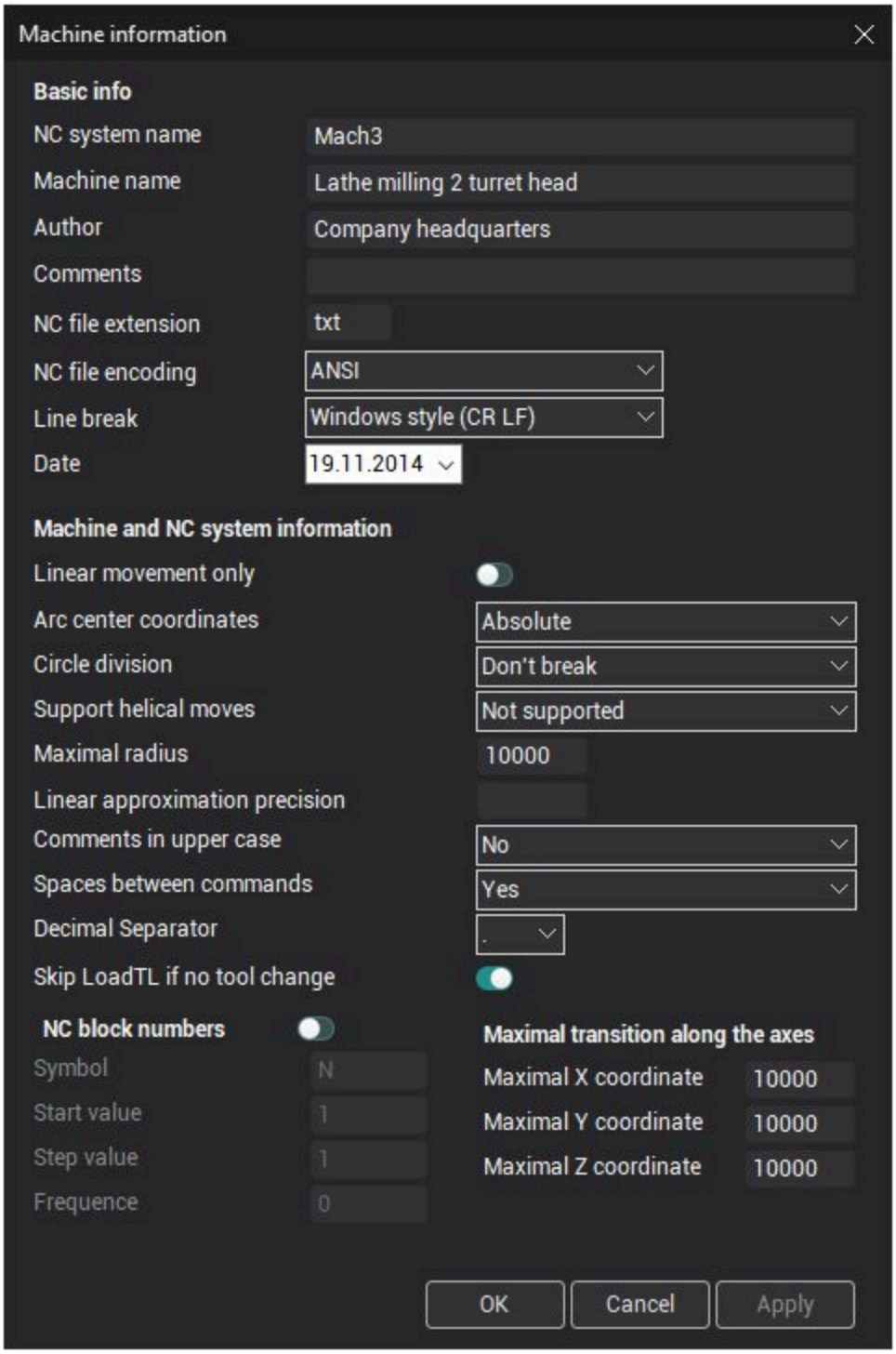
You can add a special `MakeQuery="True"` attribute to any of the nodes. Then the value for this exact parameter will be asked from the user just after opening the postprocessor file.

For example if the section has the text below:

```
<CommonDefinitions>
    <LinearApproxPrecision IsSet="False"
MakeQuery="True">0.01</LinearApproxPrecision>
```

```
        <Circles>
            <Support MakeQuery="True">True</Support>
            <HelixSupport MakeQuery="True">True</HelixSupport>
            <Center MakeQuery="True">Absolute</Center>
            <Division MakeQuery="True">DoNotBreak</Division>
            <MaxRadius MakeQuery="True">10000</MaxRadius>
        </Circles>
        <XMax MakeQuery="True">100000</XMax>
        <YMax MakeQuery="True">100000</YMax>
        <ZMax MakeQuery="True">100000</ZMax>
        <UpperCaseComments MakeQuery="True">True</UpperCaseComments>
        <Spaces MakeQuery="True">False</Spaces>
        <DecimalSeparator MakeQuery="True">Point</DecimalSeparator>
        <BlocksNumbering>
            <Address MakeQuery="True">N</Address>
            <StartValue MakeQuery="True">100</StartValue>
            <StepValue MakeQuery="True">2</StepValue>
            <Frequence MakeQuery="True">0</Frequence>
        </BlocksNumbering>
    </CommonDefinitions>
```

Then the next dialog will appear just after opening the postprocessor file (including the case of starting G-code generation from the CAM system directly):



If click Save then the data will be saved in the postprocessor file and the dialog will not appear in the next time. If click OK then the NC program will be generated with the inputted data and the dialog will appear in the next time again. If click Cancel then the postprocessor file will not be opened (G-code will not be generated).

Additional information input

Sometimes it is useful to have the system query for the list of additional parameters and user must provide the values for each of this parameter before working with the postprocessor. It is possible. The list will appear in the machine information dialog. If the postprocessor file contain the specially formed

InitialQuestions node then the "Additional information" panel with drop-down lists or edit fields appears. After the user provide some values inside these fields then the global variable corresponds to each parameter is initialized by the user input.

The **InitialQuestions** section format is shown below:

```
    <InitialQuestions>
        <!-- Multiple <Question> nodes possible  -->
        <Question>
            <QuestionSentence>User visible text for the question</QuestionSentence>
            <VarID>Name_of_variable</VarID>
            <VarType>Type_of_variable</VarType> <!-- vtString, vtReal, vtInteger
types available -->
            <VarValue>
                <![CDATA[Default value of the variable]]>
            </VarValue>
            <EnumValues>
                <!-- None or multiple <EnumValue> nodes are possible  -->
                <EnumValue>
                    <Caption>User visible caption of first possible value</Caption>
                    <Value>
                        <![CDATA[Variable first possible value]]>
                    </Value>
                </EnumValue>
                <EnumValue>
                    <Caption>User visible caption of second possible value</Caption>
                    <Value>
                        <![CDATA[Variable second possible value]]>
                    </Value>
                </EnumValue>
            </EnumValues>
        </Question>
    </InitialQuestions>
```

Where

- "User visible text for the question" – the message for the user next to the input field.
- "Name_of_variable" – the name of any global variable that is available in the postprocessor (including custom programmer's variables defined in Common handler).
- "Type_of_variable" – The type of the variable. It can be one of three values: vtString for the String type, vtReal for the Real and vtInteger for the Integer types.
- "Default value of the variable" – the start value that will be assigned to the variable.
- If the variable should have only restricted set of values (drop down list) then define a non empty "EnumValues" section.

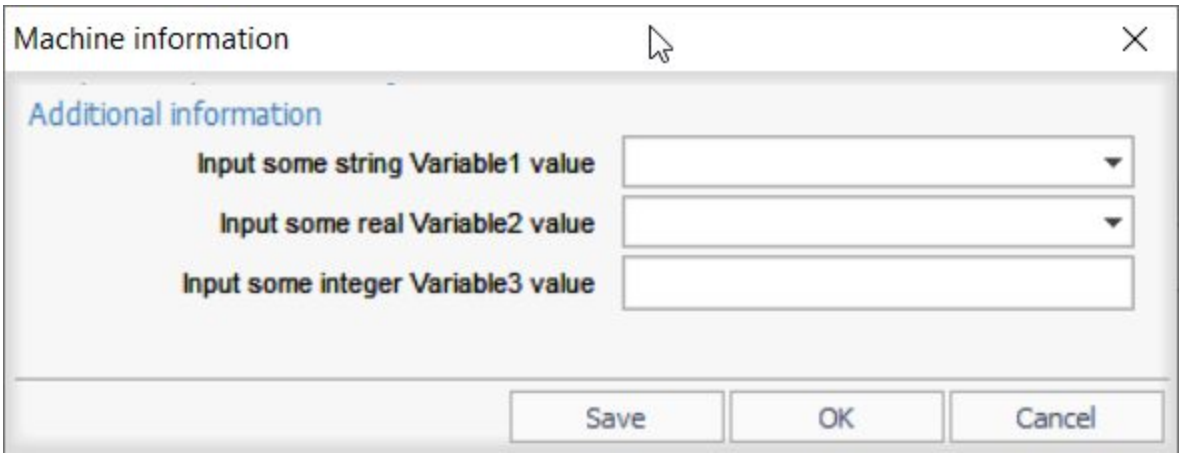
- "User visible caption of first possible value" – the user friendly value description.
- "Variable first possible value" – any value that should have the same type like the global variable.
- The text between `<!--` and `-->` is just a comment in the form of correct XML syntax.
- The syntax construction that starts with `<![CDATA[` and ends with `]]>` is required to be possible to input any characters set without breaking the correctness of XML.

An example for the InitialQuestions section is shown below:

```
<InitialQuestions>
    <Question>
        <QuestionSentence>Input some string Variable1 value</QuestionSentence>
        <VarID>Variable1</VarID>
        <VarType>vtString</VarType>
        <VarValue>
            <![CDATA[Default variable 1 value 1]]>
        </VarValue>
        <EnumValues>
            <EnumValue>
                <Caption>Value 1</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 1]]>
                </Value>
            </EnumValue>
            <EnumValue>
                <Caption>Value 2</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 2]]>
                </Value>
            </EnumValue>
            <EnumValue>
                <Caption>Value 3</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 3]]>
                </Value>
            </EnumValue>
        </EnumValues>
    </Question>
    <Question>
        <QuestionSentence>Input some real Variable2 value</QuestionSentence>
        <VarID>Variable2</VarID>
        <VarType>vtReal</VarType> <!--vtString, vtReal, vtInteger-->
        <VarValue>
            <![CDATA[10.321]]>
        </VarValue>
        <EnumValues>
```

```
                <EnumValue>
                    <Value>
                        <![CDATA[9.98]]>
                    </Value>
                </EnumValue>
                <EnumValue>
                    <Value>
                        <![CDATA[10.321]]>
                    </Value>
                </EnumValue>
                <EnumValue>
                    <Value>
                        <![CDATA[4.782]]>
                    </Value>
                </EnumValue>
            </EnumValues>
        </Question>
        <Question>
            <QuestionSentence>Input some integer Variable3 value</QuestionSentence>
            <VarID>Variable3</VarID>
            <VarType>vtInteger</VarType>
            <VarValue>
                <![CDATA[20]]>
            </VarValue>
        </Question>
    </InitialQuestions>
```

The start input dialog will be the next:



After the save button is pressed the section will be the next:

```
    <InitialQuestions>
        <Question MakeQuery="False">
            <QuestionSentence>Input some string Variable1 value</QuestionSentence>
```

```
        <VarID>Variable1</VarID>
        <VarType>vtString</VarType>
        <VarValue>
            <![CDATA[Default variable 1 value 2]]>
        </VarValue>
        <EnumValues>
            <EnumValue>
                <Caption>Value 1</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 1]]>
                </Value>
            </EnumValue>
            <EnumValue>
                <Caption>Value 2</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 2]]>
                </Value>
            </EnumValue>
            <EnumValue>
                <Caption>Value 3</Caption>
                <Value>
                    <![CDATA[Default variable 1 value 2]]>
                </Value>
            </EnumValue>
        </EnumValues>
    </Question>
    <Question MakeQuery="False">
        <QuestionSentence>Input some real Variable2 value</QuestionSentence>
        <VarID>Variable2</VarID>
        <VarType>vtReal</VarType>
        <VarValue>
            <![CDATA[9.98]]>
        </VarValue>
        <EnumValues>
            <EnumValue>
                <Caption>9.98</Caption>
                <Value>
                    <![CDATA[9.98]]>
                </Value>
            </EnumValue>
            <EnumValue>
                <Caption>10.321</Caption>
                <Value>
                    <![CDATA[10.321]]>
                </Value>
            </EnumValue>
```

```
                <EnumValue>
                    <Caption>4.782</Caption>
                    <Value>
                        <![CDATA[4.782]]>
                    </Value>
                </EnumValue>
            </EnumValues>
        </Question>
        <Question MakeQuery="False">
            <QuestionSentence>Input some integer Variable3 value</QuestionSentence>
            <VarID>Variable3</VarID>
            <VarType>vtInteger</VarType>
            <VarValue>
                <![CDATA[12345]]>
            </VarValue>
            <EnumValues/>
        </Question>
    </InitialQuestions>
```

And at the runtime the variables will have the next values:

- Variable1 = "Default variable 1 value 2";
- Variable2 = 9.98;
- Variable3 = 12345;

See also

The main window

Postprocessor masks

Use masks to specify the NC-block templates for the given CLData command.

Each CLData command has its own mask. Masks use a special language that makes them similar to NC-program blocks.

For example suppose the CLData is:

```
GOTO.abs X 19.599, Y -73.200, Z 28.030
```

suppose we need to form the following NC-code to process the command:

```
G1 X19.599 Y-73.200 Z28.030 F200
```

The mask that forms the desired NC-code is:

```
G[1] X[XT] Y[YT] Z[ZT] F[FT]
```

where:

- XT, YT, ZT – current coordinates of the cutter;
- 1 – interpolation mode (positioning G0, linear G1 or arc G2, G3 (this can be different for non G-Code NC));
- FT – feedrate value.

Use variables and register values, the built-in GMA array, built-in functions and expressions that result in numerical values in the masks. Also you can use special words (switches and modifiers) that control the output of the code into the NC-program.

See also

Mask structure

Mask switches

Mask modifiers

The transformation of a mask to the subprogram

The interactive to create the masks

Mask structure

Mask can contain some lines. In general, the mask line looks like:

```
element1 element2... elementN.
```

See also

Postprocessor masks

Mask element

Registers in the masks

Modifiers

Expressions

Nested mask

The separators of the mask elements

Variables assignment from the mask

Deferred masks

GMA array in masks

Mask element

Any mask element is output to the NC-program line in the form as it looks in the mask. For example:

- Mask:

```
G0 rapid movement
```

- NC code:

```
G0 rapid movement
```

If the mask element is in a brackets "[" and "]" then it is replaced by value or variable that located in the brackets. For example:

- Mask:

```
[XT] [200]
```

- NC code:

```
100.12456 200
```

In the sample, the value of the XT variable is equal to 100.12456.

In the brackets, it is possible to use:

- All variables of the COMMON subprogram.
- Predefined (reserved) variables: XT, YT, ZT, XC, YC, ZC, INTERP, TOOLRAD, CLDATA$, ARCPLANE, XP, YP, ZP, FEED, TLCOMP, TRCOMP, FROMX, FROMY, FROMZ, CURCODE, NCNAME$, NCPATH$, BLOCKSTEP.
- Predefined (reserved) functions: FLAGIN, CROSS, NEXTTOOLNUM, CURDATE, CURTIME.
- All parameters of the current technological command passed by the CLD array.

Numbers have the different representation in the NC-program. The number out method is defined by the next parameters:

- Decimal point – this field can have following values:
  - Is absent,
  - Is present if the number has the fractional part,

    - Is present anyway;
- Integer width – the maximal digits number to represent the integer part of the number;
- fractional width – the digits number to represent the fractional part of the number;
- Leading zeroes and Non-significant zeroes – defines the zeroes output mode before and after the number;
- Sign – defines the output mode for the sign of the number. Following options are available:
    - No,
    - "-" only,
    - "+" only,
    - "+" and "-" always.

When the value is output to the NC-program block, the default number format is used:

- Leading zeroes and Non-significant zeroes – is absent;
- Decimal point – is present if the number has the fractional part;
- Sign – is present if the number is negative;
- Integer width and Fractional width – allows to out number without rounding
- Element identifier is output as text.

See also

Mask structure

Registers in the masks

Registers are used to define the format of the number output according to the requirements of the NC systems. The register in the postprocessor unites the next properties:

- Identifier of the register;
- Current value of the register;
- Previous value;
- Value output format;
- The name of the register.

See section The block structure and format definition for more details.

If write the register name or register identifier before the open square bracket then the number output format of this register will be used for the value output. For example:

- Mask:

```
G_INTERP[1] X[XT] Y[YT] Z[ZT] F[200]
```

- NC code:

```
G1 X100.100 Y-245.100 Z-010.560 F200
```

While the mask analysis it is presumed that the word before the open square bracket defines the register. Executable system looks for the register by name, if the register is not found then the system looks by identifier. The registers are looked in order how it has defined in the registers list. So if the list has some registers with the same identifier name then the first will be used.

If register is not found then value is output by default number format and the word that is in mask before the open square bracket is output before the number. For example:

- Mask:

```
G_INTERP[1] X[XT] YYY[YT] Z[ZT] F[200]
```

- NC code:

```
G1 X100.100 YYY-245.100034 Z-010.560 F200
```

In the sample the word YYY is absent in the registers list.

If the register is found then the current register value is assigned to the old register value and the new value is assigned to the current register value. After that, the executable system compares the old and the current values and if they are not equal then the register identifier and value is output to the NC-program line accordingly to the defined number format.

Example. Lets the current register F value is equal to 200:

- Mask:

```
G_INTERP[1] X[XT] Y[YT] Z[ZT] F[200]
```

- NC code:

```
G1 X100.100 Y-245.100 Z-010.560
```

The old and new value of the register is equal, so the register F value is not output to the NC code.

See also

Mask structure

Modifiers

Sometimes, it is required to output the new register value independently if it is changed or not, and conversely to change the register value and do not output it to the current line. The modifiers On and Off are intended to do it.

The modifier is written in the square brackets after the value and comma.

Modifier On. If the modifier is written then the register value is output to the NC code in any case.

For example, lets the current register `G_PLANE` value is equal to 17:

- Mask:

```
G_PLANE[17, On] X[XT] Y[YT]
```

- NC code:

```
G17 X100.123 Y200.456
```

It is possible to out current value of register without the new value assign. To do it it is necessary to write On without value. For example:

- Mask:

```
G_PLANE[On] X[CLD.X] Y[CLD.Y]
```

- NC code:

```
G17 X100.123 Y200.456
```

Modifier Off. If the modifier is written then the new value is assigned to the register but the register is not output to the NC code. For example:

- Mask:

```
G49G80M5M9 G_PLANE[17,Off]
```

- NC code:

```
G49G80M5M9
```

It is possible to use modifier Off without value. It is needed to exclude the out of the current register value to the NC code without the assigning of a new value.

See also

Mask structure

Expressions

It is possible to write the expressions to the masks. The expression is the mathematical formula. The result of expression is a number.

Syntactically, the math expression is the combination of the numbers, numerical variables and numerical functions, separated by the signs of the math operations and the parentheses. The simplest examples of math expressions are the number and the numerical variable

Following math operations are allowed in the language:

- `+` – the addition;
- `-` – the subtraction;
- `*` – the multiplication;
- `/` – the division;
- `^` – the exponentiation.

It is necessary to remember, that two operational sign can't follow one-after-another.

Following math functions are allowed:

- `SIN(x)` – sine of angle x (in degrees);
- `COS(x)` – cosine of angle x (in degrees);
- `TAN(x)` – tangent of angle x (in degrees);
- `ATN(x)` – arctangent of angle x (in degrees);
- `ASIN(x)` – arcsine of angle x (in degrees);
- `ACOS(x)` – arccosine of angle x (in degrees);
- `SQR(x)` – square root of x;
- `ABS(x)` – absolute value of x;
- `SGN(x)` – sign for x;
- `ROUND(x, y)` – rounding of x to y decimals after point;
- `LOG(x)` – decimal logarithm of x;
- `LN(x)` – natural logarithm of x.

Predefined variables and functions, all parameters of current CLDATA command transmitted via the CLD array are available in the expression. For example:

- Mask:

```
X[2+5/2+2*(sin(45))] Y[CLD.Y*2] F[FEED]
```

- NC code:

```
X005914 Y001234 F200
```

In this sample the expression of the first element: 2+5/2+2*(sin(45)) is equal to 5.91421356237697. When out to the NC code, the X register was found in the registers list and the number output format was taken from the register. The expression of the second element has the parameter transmitted via the CLD array – `CLD.Y` (equal 0.617) multiplied on 2. The third element uses the predefined variable FEED. This variable contains the current feedrate value.

See also

Mask structure

Nested mask

Sometimes it is required to out the register value if the value of other register is changed. Then the nested mask is used. For example:

- Mask:

```
Z[CLD.Z, G[43]]
```

- NC code:

```
G43 Z0.123
```

The value of the register Z is changed and value 43 is assigned to the register G.

It is possible to use the modifiers, expressions, and other nested masks in the nested masks. The registers of the nested mask will be included in to the NC-program line in the order according to the registers list. For example:

- Mask:

```
X[CLD.X] Y[CLD.Y] Z[CLD.Z, G[43]]
```

- NC code:

```
G43 X6.141 Y-4.234 Z0.123
```

The changed register G is located in the start of line according to the registers list.

See also

Mask structure

The separators of the mask elements

In general, the mask line consists of some elements. It is possible to insert the spaces between the elements. The spaces has not influence on the NC-program line.

To control the spaces between the words in the line it is possible to use the parameter Spaces between commands. If the parameter is checked then the spaces is added even if the separator is absent between the elements. For example:

- Mask:

```
" G[2] X[100.101]Y[234.89] Z[45.67]"
```

- NC code:

```
" G2 X100.101 Y243.890 Z045.670"
```

In the line, the space is added between the first and the second elements. The spaces quantity before the first between two and three, three and four is not changed.

If parameter Spaces between commands is off then all spaces between elements will be removed independently of it is quantity. For example:

- Mask:

```
G_INTERP[INTERP] X[CLD.X] Y[CLD.Y] Z[CLD.Z]
```

- NC code:

```
G1X-49.47Y-6.513Z.033
```

All spaces is removed from the NC-program line.

See also

Mask structure

Variables assignment from the mask

It is possible if necessary to assign any predefined variables or COMMON variables from the mask. Values can be assigned fro the number and string variables. The constants only can be assigned to the string variable. Any expression including functions and other variables can be assigned to the number variable.

The variable values can be assigned in any line of the mask. NC program line is not generated for this mask line. The `;` sign is the delimiter between the variables. For example:

- Mask:

```
INTERP=0;XP=XT;YP=YT-Y1+2;CLDATA$="text"
G_INTERP[INTERP] X[XT] Y[YT] Z[ZT] ([CLDATA$])
```

- NC code:

```
G0 X100.122 Y231.567 Z010.546 (text)
```

See also

Mask structure

Deferred masks

Sometimes it is necessary to use the current parameters of the current command when other command is processed. There are two ways to solve this task. The first way is to save the parameters to the temporary variables and use it later in the required command. The second best way is the using of the deferred masks. The current parameters of these masks are saved and the mask execution is deferred to the required program execution. When the deferred mask is executed, the current parameters are replaced by the saved parameters.

The deferred mask must be in the braces `{` and `}`. Inside the braces necessarily to define one of the CLDATA commands. The mask is executed when this command appears. The lines before this command executed before command execution and the lines placed after the defined command are executed after the command execution. For example:

- Mask of command SafPos:

```
{
G_INTERP[0] X[XT] Y[YT] Z[CLD.Z]
G_FUNC[28] X[CLD.X] Y[CLD.Y]
LoadTL
G_INTERP[1] G_FUNC[29] G_LENGTHCOMPENS[43] X[XT] Y[YT]
G_LENGTHCOMPENS[49] Z[ZT]
}
```

- Mask of command LoadTL:

```
H[CLD.N] T[CLD.N]
```

- Mask of command AbsMov:

```
X[CLD.X] Y[CLD.Y] Z[CLD.Z]
```

- CLData commands:

```
GOTO.abs X 20.0000,Y 20.0000,Z 20.0000
SAFPOS X 10.0000,Y 11.0000,Z 12.0000,N 0
LOADTL N 1,X 0.0000,Y 0.0000,Z 0.0000,D 1.0000,M 0,K 0,L 0.0000,
P 0.0000,A 0.0000,R 0.0000,PLANE XY(33),Dur 0.0000
```

- NC code:

```
X20.0000 Y20.0000 Z20.0000
G0 Z0.0000
G28 X0.0000 Y0.0000
H1 T1
G1 G29 G43 X20.0000 Y20.0000
G49 Z20.0000
```

In this case, when the SafPos command is executed then the deferred mask is created. The values of `CLD.X`, `CLD.Y`, `CLD.Z` of SafPos command are saved and the mask execution is deferred to the LoadTL processing. Values of XT, YT, ZT is not saved because it is not the CLData command parameters. Before the LoadTL processing the mask below is executed:

```
G_INTERP[0] X[XT] Y[YT] Z[CLD.Z]
G_FUNC[28] X[CLD.X] Y[CLD.Y]
```

After that, executed:

```
LoadTL: H[CLD.N] T[CLD.N]
```

Finally executed:

```
G_INTERP[1] G_FUNC[29] G_LENGTHCOMPENS[43] X[XT] Y[YT]
G_LENGTHCOMPENS[49] Z[ZT]
```

If write the Keep identifier before the right brace then deferred mask will be always performed then the required program is executed (It is LoadTL in the previous sample). For example:

- Mask of command SafPos:

```
{
G_INTERP[0] X[XT] Y[YT] Z[CLD.Z]
G_FUNC[28] X[CLD.X] Y[CLD.Y]
LoadTL
G_INTERP[1] G_FUNC[29] G_LENGTHCOMPENS[43] X[XT] Y[YT]
G_LENGTHCOMPENS[49] Z[ZT]
Keep
}
```

In this sample, the execution of the deferred mask will be performed every time when the LoadTL is processed.

See also

[Mask structure]

GMA array in masks

The predefined GMA array is used to conveniently program the MULTIGOTO command postprocessor either as a technology command program or as a mask. The GMA array like the CLD array is initialized before the command is processed. Every item of the GMA array corresponds to an axis in the machine scheme.

The GMA array usage syntax in masks is different from the postprocessor program's code: when used in a mask the "GMA" keyword and brackets are omitted. To access a GMA array item's property when programming a mask for the MULTIGOTO command use the quoted name of the axis and the property name without quotes separated by a point. For example to put the current value of the coordinate named AxisXPos into a NC-program the mask `X["AxisXPos".Vn]` should be used.

The following are examples of accessing GMA arrays items:

- `"<AxisName>".OutFlag` – whether the coordinate is present in current MULTIGOTO command.
- `"<AxisName>".Vn` – current value of the coordinate.
- `"<AxisName>".Vp` – previous value of the coordinate.
- `"<AxisName>".Axis` – coordinate name in the machine scheme.
- `"<AxisName>".Reg` – name of the register bound to the coordinate.
- `"<AxisName>".TurnCount` – number of full revolutions (360 degrees) of the current movement of rotatable axis.
- `"<AxisName>".Dir` – direction of rotation.

AxisName – name of the machine coordinate.

The following example writes the X, Y and Z coordinates values into the NC-program passed through the MULTIGOTO command for the axes AxisXPos, AxisYPos and AxisZPos respectively.

```
X["AxisXPos".Vn] Y["AxisYPos".Vn] Z["AxisZPos".Vn]
```

See also

Mask structure

Mask management

No content in this page. See child topics

Mask switches

Any mask is linked with the three switches:

1. Out previous. If the switch is checked then the NC-line is formed before the mask analyzing. All registers in which the old and current values are not equal will be out to the line.
2. Poll all registers. If the switch is checked then all registers in which the old and current values are not equal will be included in to the line in addition to the registers of the mask. If such registers is present (it is possible if the Off modifier was used in the previous mask) then this register will be located in the line according to the order in list of registers. For example:

- Registers list:

```
G_INTERP, G_PLANE, X, Y, Z
```

- Plane mask:

```
G_PLANE[ARCPLANE,Off]
```

- AbsMov mask:

```
G_INTERP[1] X[CLD.X] Y[CLD.Y] Z[CLD.Z]
```

- CLData commands:

```
PLANE XY(33)
GOTO.abs X100 Y100 Z100
```

- NC code:

```
G1 G17 X100.000 Y100.000 Z100.000
```

3. Out block. If the switch is not checked then all registers written in the mask will be modified but the NC-program line is not out.

See also

Postprocessor masks

The transformation of a mask to the subprogram

The situation is possible then the mask possibility is not enough. To use more powerful and flexible subprograms without manual mask programming, there is the function Transform the mask to the subprogram. Use the `M -> Prg` button to transform the mask. The button is located near the mask switches.

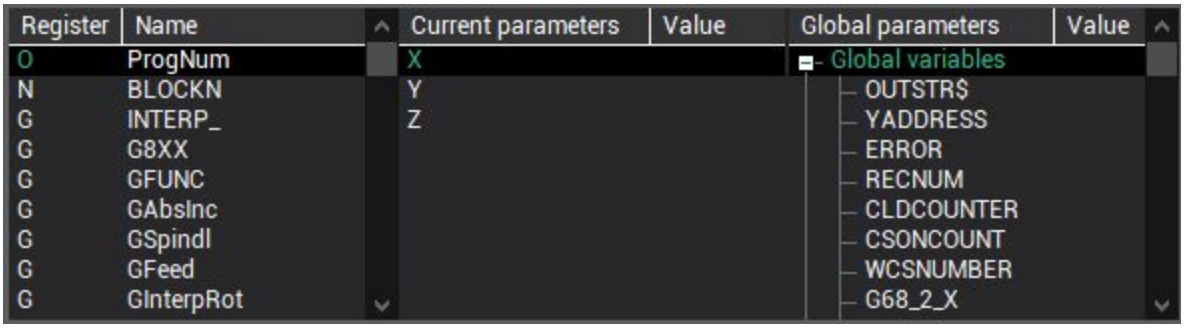
To undo the transformation it is necessary to move by clipboard the mask line from the subprogram editor to the mask editor and to set the switches.

See also

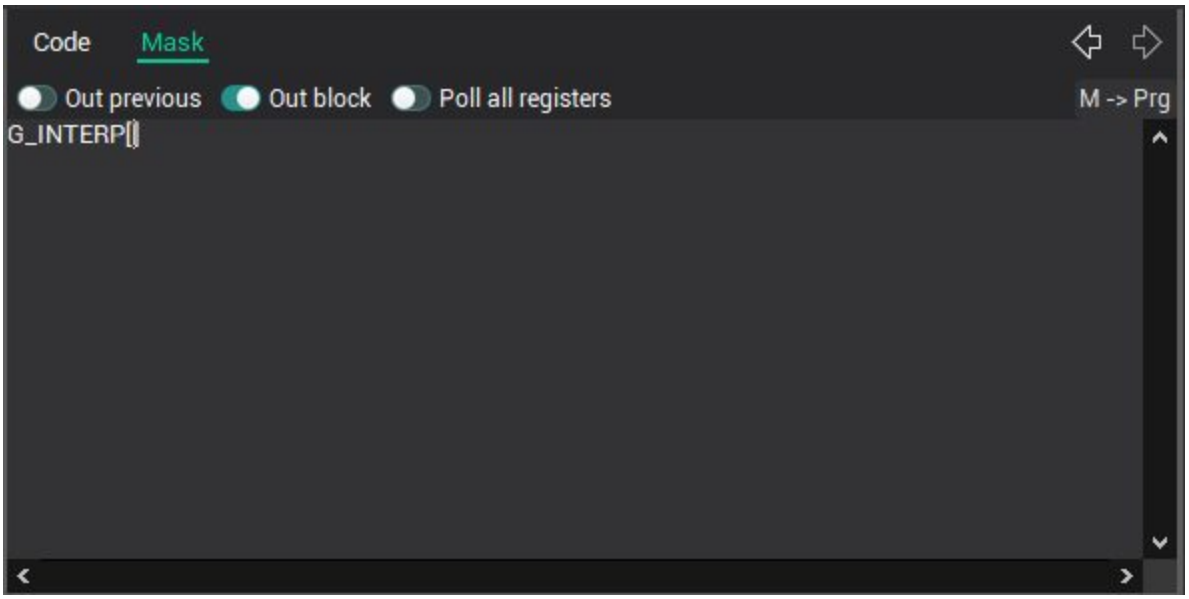
Postprocessor masks

The interactive to create the masks

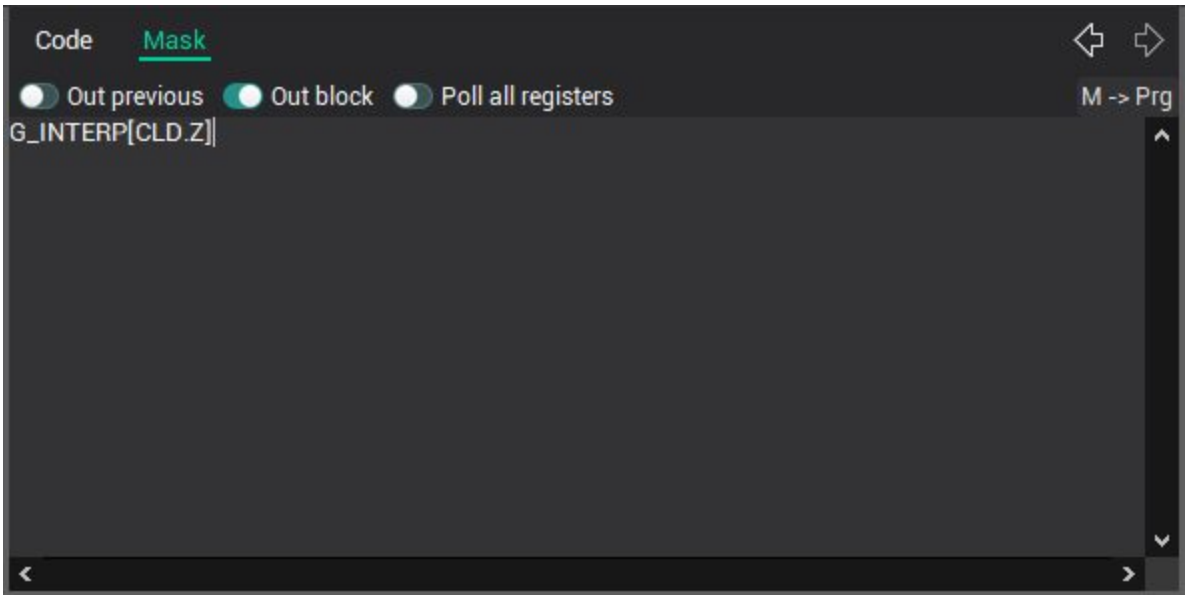
There are the three lists for the comfortable masks editing. These lists are placed under the mask editor. The left list is the list of the registers. The second list is the list of the current CLData command parameters. The right list is the list of the global variables and predefined functions.



The double click on the registers list inserts the selected register name to the cursor position. The chars `[]` added after the register name and the cursor moves to the position between.



The double click on the current parameters list inserts the CLData parameter name to the cursor position. Prefix CLD is added to the parameter.



See also

[Postprocessor masks]

Language description

No content in this page. See child topics

Basic definitions

No content in this page. See child topics

Conditional indications

The following rules and conditional indications are used in this document to describe the statements formats:

- Capital characters are used to indicate the reserved words;
- Line characters are used to indicate the names, the variables, numerical or arithmetical expressions;
- Linguistic expressions are indicated by . . symbols;
- Unnecessary parts of the operators are indicated by {..} symbols;
- Alternative parts of the operators are divided by | symbol.

Technology command programs, comments

Postprocessor generates NC-program code using the files of technology commands (CLData). Briefly the process of postprocessing can be outlined as following:

1. A command is read from the file.
2. The command is analyzed, it's code and properties are determined and the built-in CLD array and predefined variables are initialized.
3. If a technology command program is enabled in the postprocessor and it's code is defined then it is called by the system. The technology command should contain the code to form the NC-program lines corresponding to the CLData.
4. If a mask is enabled and specified then the mask is used to form the NC-program code.

Technology command programs are coded on the task-oriented language and consist of mathematical expressions and functions, input/output operators, program flow control operators, subprogram call's, NC-program code output operators.

Technology command program starts with the PROGRAM keyword followed by the program name, the program ends with the END keyword. The name of the program is the same as the name of the CLData command which it processes. Technology command parameters are passed through the built-in CLD array and special Cmd operator. The detailed description of CLData commands and parameter values is found in the Technology commands description chapter.

Program code consists of operators and comments. It is possible to write multiple operators in a single line of code if you separate them with the `;` symbol. It is legal to carry operator code over a line if it's parts are logically incomplete.

Comments are useful to provide insight over some more complicated parts of your program. To place a comment type the `!` symbol and all the text after that symbol will be marked as a comment and thus ignored by the compiler. To comment/uncomment several lines, select the appropriate lines and press `Ctrl+!` or `Ctrl+/`.

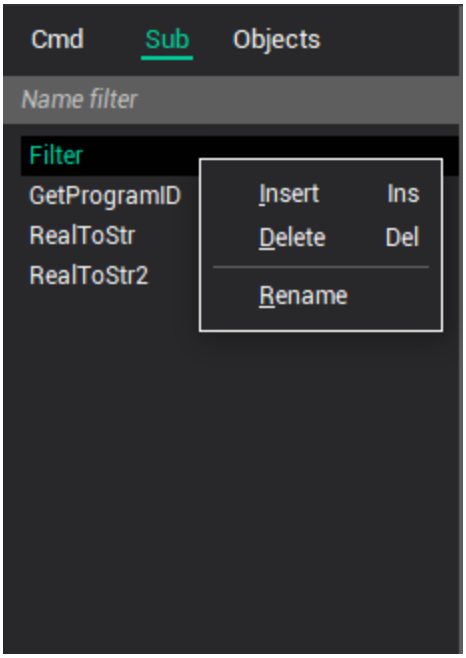
It is also possible to comment out any part of the text by enclosing it in a pair of curly braces `{` - `}`. This way of commenting allows you to comment both an arbitrary part of a line and several consecutive lines, including lines commented out with the `!` symbol.

In the course of postprocessor execution the first technology program to be called is the COMMON program. Then the PARTNO program is executed, the FINI program is always the last.

The COMMON program is used to define global variables, that is all variables defined in the COMMON program are accessible in every part of the postprocessor. For their own needs, any program can use variables, not declared in COMMON and, therefore, are not publicly available. Such variables are called local.

Subprograms

Subprograms tab holds the list of user defined subprograms and the Filter subprogram. Use the context menu to add, remove or rename subprograms. The code of the subprogram can be viewed or edited in the code editor. The code editor displays the code of the selected subprogram.



The name of the subprogram can be any legal string identifier, provided it is not the name of a technology program or already used by other subprogram.

The principles of definition and syntax of subprograms is described in the Language description.Subprograms topic.

See also

The main window

The language statement conception

The statement is a functional unit of the language. The statement formats are defined uniquely by the set of the available linguistic constructions and definition methods. The examples of the statement on the postprocessor language are:

```
PRINT "symbol" ! prints the literal string "symbol"
JUMP 1 ! jumps to the label 1
```

The operators of the language can contain the identifiers, numbers, literal strings and auxiliary symbols.

The set of symbols

The following symbols are used for language statements designing:

- `a..z`, `A..Z` – capital and lowercase letters of Latin alphabet;
- `0..9` – numerals from 0 to 9;
- `_` – the underlining;
- – the space;
- `,` – the comma;
- `=` – the equal;
- `;` – the semicolon;
- `:` – the colon;
- `[` – left bracket;
- `]` – right bracket;
- `"` – double quotation marks;
- `(` – left parenthesis;
- `)` – right parenthesis;
- `+` – the plus;
- `-` – the minus;
- `*` – the asterisk;
- `/` – the slash;
- `\` – the backslash;
- `^` – the caret;
- `$` – the dollar;
- `<` – left angular bracket;
- `>` – right angular bracket;
- `#` – not equal;
- `.` – the point;
- `@` – commercial A.

The comments and literal strings allow any symbol.

The Identifier – is a unceasing letters sequence, numerals and symbols `_`, `@` and `$`, and must begin from the letter. Capital and lowercase letters are indistinguishable in the identifiers.

The Number – is an unceasing sequence of numerals, containing only one decimal point. The signs `+` or `-` is allowed before the number. As default, the number is interpreted as positive.

The Literal strings – are the arbitrary symbols sequences, marked by the double quotation marks.

The variables

Variable – a named memory location used to refer data at that location. In other words, variable - the name, which may be due to a certain value.

Variables can have different types. The type of a variable defines the set of values that can be stored in the variable and the set of operations that can be performed on that variable. Variables in the Postprocessors generator can be one of the following types:

- Integer (integer number) – use Integer to represent an integer number. The Integer variable value range from -2147483648 to 2147483647.
- Real (real number) – use Real to store numbers with fractional values. The range of values that can be represented by the Real type variable is: 5.0*10^-324 .. 1.7*10^308. The Real type has 15-16 significant digits.
- String (character strings) – use String to store a sequence of up to ~2^31 characters.
- Array (arrays or vectors) – a complex type that represents a sequence of items of simple type values.

Variables can be local scope or global. Local scope variables are accessible only in the current program or subprogram. Global variables can be accessed in any program or sub of the postprocessor. To define a global variable define it in the special COMMON program. All variables defined in the COMMON program are global. It is allowed to declare several variables of the same type in one line.

The Postprocessor generator language allows definition of variables anywhere in the code of a program or a subprogram, but it is a good practice to define the variables used in the program at the program head. Use the following syntax to define a variable:

`Variable name 1`, {`Variable name 2`}: Type of the variable

Here:

- Variable name – a unique in the current scope identifier of the variable – an identifier must begin with an alphabetic character or an underscore (_) and cannot contain spaces; alphanumeric characters, digits, underscores, and symbols `@` and `$` are allowed after the first character. The identifiers are case insensitive. Reserved words, operator names, technology command program names and subprogram names cannot be used as identifiers.
- Type of the variable – one of the types: Integer, Real, String, Array.

The following is an example of variable definition and usage.

Example

```
sub Sub1
  nn: Integer ! Integer type variable definition
```

```
  xx, yy, zz: Real ! Real type variables definition
  ss: String ! String type variable definition
  nn = 2 ! Variable value assignment
  xx = 7.43 ! Variable value assignment
  yy = 12.6 ! Variable value assignment
  rr = nn * sqr(xx^2 + yy^2) ! variables in expression
  ss = "Result = " ! Variable assignment
  print ss, rr ! debug output value of the variable
subend
```

Arrays

An array represents an indexed collection of elements of the same type. The array is characterized by its name, the type of stored items, the size (number of stored items) and the numbering of elements. An array can be viewed as a table:

Array item index	Array item value
1	Value1
2	Value2
...	...
N	ValueN

In the table above N – an array size (number of items in the array). There is no limitation to the maximum array size, although using exceptionally large arrays is not recommended. All arrays are indexed from 1 up to array size.

Postprocessor language supports both one-dimensional and two-dimensional arrays. Array items can be of any simple type: (Integer), (Real), (String) or (Record).

Arrays can be used anywhere where simple variables of the same base type are legal. To access array item you need to specify array name and item index, index can be any numeric expression that doesn't violate the range bounds:

```
<ArrayIdentifier>[<ItemIndex>, {<ItemIndex>}]
```

To define an array variable use the following syntax:

Array variable name: array Array size of Array base type

Where

- Array variable name – any legal identifier;
- Array size – positive integer number;
- Array base type – one of simple types (Integer, Real or String).

To define two-dimensional array use next syntax:

Array variable name: array [First array size, Second array size] of Array base type

Example

```
V: array 10 of Real ! Definition of an array of 10 real items
V[5] = 10.67 ! Assigning value to the fifth array item
i = 7
! Using array in expressions
i = 7
V[10] = 15*sqr((sin(V[i]))^2 + (cos(V[i+1]))^2)


V: array [3, 3] of Real ! Definition of two-dimensional array

for i = 1 to 3 do for j = 1 to 3 do V[i, j] = i*10+j
```

An array with hardcoded number of elements is called a static array. An array whose size can change along the course of program execution is called dynamic array. To define a dynamic array omit the array size in the array definition. Dynamic arrays are indexed from 1. The size of dynamic array is set to the maximum item index that was accessed.

The following is an example of string array definition and usage.

Example

```
ArrayS: array of string ! Array definition
! Fill array items
ArrayS[1] = "Hello," ! There is no need to specify
ArrayS[2] = " World" ! the number of items
ArrayS[3] = "!" ! the memory is allocated automatically
! Array use
output ArrayS[1] + ArrayS[2] + ArrayS[3]
```

Mathematical expressions and functions

Mathematical expression is the exact operations description, which operations produce numerical value. They are the analogs of mathematical formulas.

Syntactically, the math expression is the combination of the numbers, numerical variables and numerical functions, separated by the signs of the math operations and the parentheses. The simplest examples of math expressions are the number and the numerical variable. Following math operations are allowed in the language:

- `+` – the addition;
- `-` – the subtraction;
- `*` – the multiplication;
- `/` – the division;
- `^` – the involution.

It is necessary to remember, that two operational sign cannot be placed one after another. To control the order of evaluation use the parenthesis `(` and `)`.

Also you can use predefined functions and variables in the expressions.

Predefined variables and functions

Postprocessor language offers the following built-in and predefined functions and variables:

- Standard mathematical functions;
- String functions;
- Miscellaneous functions and variables;
- CLData functions and operators.

Standard mathematical functions

The following mathematical functions are implemented:

- `SIN(x)` – sine of angle x (degrees).
- `COS(x)` – cosine of angle x (degrees).
- `TAN(x)` – tangent of angle x (degrees).
- `ATN(x)` – arctangent of x, returns in degrees.
- `ASIN(x)` – arcsine of x, returns in degrees.
- `ACOS(x)` – arccosine of x, returns in degrees.
- `SQR(x)` – square root of x.
- `ABS(x)` – modulus of x.
- `SGN(x)` – sign of x. Returns "-1", if x is less than zero; returns "+1", if x is greater than zero; returns "0", if x is zero.
- `ROUND(x, y)` – returns the rounded value x with y decimals.
- `LOG(x, y)` – returns the logarithm of y to base x.
- `LG(x)` – returns the common logarithm of x.
- `LN(x)` – returns the natural logarithm of x.

Here the x and y are the arguments of a function. They can be a numeric literal constant, variable, function or any expression that results in numeric value.

See also

Predefined variables and functions

String functions

The following string functions are implemented:

- `CHR(x)` – returns the character for a specified ASCII value x.
- `ORD(Str)` – returns the ordinal value of the first character of the string Str.
- `LEN(Str)` – returns the length of the string Str (the number of characters in the string).
- `POS(Pat, Str)` – returns the index of the first occurrence of the string Pat in the Str string. Returns 0 if the Pat string is not found in the Str string.
- `NUM(Str)` – converts a string Str to a numeric representation; If the string is invalid, the function returns 0.
- `STR(n)` – converts number n into a string representation.
- `COPY(Str, n, m)` – returns a substring of Str containing m characters and starting at index n.
- `UPCASE(Str)` – returns a copy of the string Str in upper case.
- `REPLACE(<string variable>, <looked string>, <string for replace>)` The replace statement of the substring in a string REPLACE.

Where

- Str, Pat – expressions that evaluate into a string value (variables, literal constants, functions or expressions);
- n, m – numeric values.

See also

Predefined variables and functions

The replace statement of the substring in a string REPLACE

Miscellaneous functions and variables

Postprocessor language provides various miscellaneous functions and variables which are used to simplify the postprocessor developing.

- CLData functions and operators.
- Registers. There is a variable for every register in the postprocessor, which name matches the register name. This variable holds the current value of the register. Register variables are of Real type.
- Previous register values. A variable which name matches the name of register with the @ symbol at the end is used to hold the previous value of the register. For example, X@ – previous value of the X register. This variable is initialized then the NC-block is output into the NC-program.
- RGS array. The RGS array allows you to get access to the registers by their index in the list. It can be useful in some cases (for example inside multichannel postprocessors where you can have separate set of the registers for each channel and you need to switch quickly between them) instead of using name of the register.
- CLD array. The array CLD is an array of real numbers, it is initialized by the system with the current CLData command parameters.
- RecNum returns the number of parameters of the current technology command which are filled in the CLD array. The index of the top element of the CLD array is equal to RecNum.
- GMA array. Use it to process the parameters of the MULTIGOTO technology command.
- CLData$ variable. This variable is initialized by the system with a textual data of the technology command if the command parameter is a string of text. CLData$ holds the passed value until it is reset by other text technology command. The type of the CLData$ is String.
- OutStr$ variable. This variable is assigned the current NC-code line after the execution of FORMBLOCK or OUTBLOCK command. OutStr$ is String.
- BlockStep variable is assigned the value of increment when the automatic NC-code line numeration is used. Automatic NC-block numeration is enabled in the Machine information dialog. If the auto numeration is enabled then when each new NC-block is formed it's number is increased by the BlockStep. If you set BlockStep to 0 then the automatic NC-block numeration will be disabled.
- String variable NCName$ is assigned the name of the file of the NC-program (no file path and no file extension).

- String variable NCPath$ is assigned the full NC-program file name including file path and file extension.
- String variable SPPName$ holds the name of the postprocessor file without path and extension.
- String variable SPPPath$ holds the full file path of the postprocessor file including file extension.
- CurDate – function returns the string representing the current date.
- CurTime – function returns the string representing the current time.
- CurISODate – function returns the string representing the current date and time in ISO 8601 format.
- Error variable – integer variable which value is set to the last error code if an error occurs during program execution.
- Xt, Yt, Zt real variables – current tool position coordinates. These variables are initialized by the system directly before the commands CIRCLE, FROM, ABSMOV, GOHOME are processed.
- Xp, Yp, Zp real variables – previous tool position coordinates. These variables are initialized by the system directly before the commands CIRCLE, FROM, ABSMOV, GOHOME are processed.
- Xc, Yc, Zc variables – circle center coordinates (initialized by the system prior to the CIRCLE program call).
- Interp – current interpolation mode (the value of this variable is assigned automatically before the CIRCLE, FEDRAT, ABSMOV, RAPID, GOHOME programs are executed). The set of values for the Interp variable:
    - 0 – rapid feedrate,
    - 1 – linear interpolation,
    - 2 – clockwise circle interpolation,
    - 3 – counter clockwise circle interpolation.
- ToolRad – tool radius (initialized by the system prior to the LOADTL execution).
- ArcPlane – current plane of circle interpolation (the variable is assigned it's value directly before the call of the PLANE program), the value of this variable can be one of the following:
    - 33 – XY,
    - 37 – YZ,
    - 41 – XZ,
    - 133 – YX,
    - 137 – ZY,
    - 141 – ZX.
- Feed – current feedrate value (the variable is set prior to FEDRAT call).

- TlComp, TrComp variables – length and radius compensation values respectively (set before the CUTCOM and LOADTL commands execution).
- FromX, FromY, FromZ variables – coordinates of start point (set before the FROM command).
- FlagIn returns:
  - 1 – if the next intersection is passed from the inside;
  - 0 – if the next intersection is passed from the outside;
  - -1 – if the intersection kind cannot be determined (for example, two adjacent NC-blocks form a colinear cuts for linear interpolation).
- Cross – function returns 0, if the current toolpath element conjugates with the next element, 1 is returned if the elements intersect. Generally two cuts always intersect, cut and arc can conjugate or intersect. This function can be used to implement compensation.
- NextToolNum{(Mode)} – returns the number of the next tool. Optional parameter Mode defines behavior for the last tool change. If mode is 0 (default value when no parameters defined) then it returns -1 after last tool change command. If mode is 1 then it returns the number of the first tool in the project.
- ToolChange – returns the indication of tool change (whether there will be LOADTL command in the CLData commands list):
  - 1 – there were no tool change but there will be tool change in the following commands or tool change command is being currently processed;
  - 0 – there were a tool change or it is being processed in the current command;
  - -1 – the current tool change is the last one in the CLData commands list.

The table below illustrates the way ToolChange function works.

List of CLData commands	**ToolChange** function return value
PARTNO	1
...	1
...	1
...	1
LOADTL	1
...	0

List of CLData commands	**ToolChange** function return value
...	0
...	0
LOADTL	0
...	0
...	0
...	0
LOADTL	-1
...	-1
...	-1
...	-1
FINI	-1

- WorkingMode – returns the working mode of PostProcessor Generator:
  - 1 – the PostProcessor Generator is started by CAM system during the operation, while the "G-code based simulation" mode is on. If all operations are executed at once in CAM system, PARTNO and FINI are the first and last technological commands, respectively. If the operations are performed by ones, the PARTNO and FINI commands do not generate;
  - 0 – PostProcessor Generator is run as a separate application or on the generating of the NC program from CAM system on the Machining tab.

See also

Predefined variables and functions

GMA array

Records

`Record` is a keyword used to describe variables - structures. The syntax for declaring a variable of type `Record` is:

```
RecVarName: Record
  RecFieldName1: Type1
  RecFieldName2: Type2
  .....
  RecFieldNameN: TypeN
End
```

Where

- RecVarName – name of variable;
- RecFieldName1...RecFieldNamen – variable field names;
- TypeX – record field type. It can be any simple type, record or static array of the above types.

Example

```
Rec1, Rec2: record
  X, Y, Z: real
  P: array [3,3] of integer;
  S1, S2: string
end
RR: array [10] of record
  X, Y, Z: real
end
Rec11: record
  field1: integer
  field2: array 10 of string
  field3: record
    nestedField1: real
    nestedField2: array 5 of integer
  end
end
```

Using Records

Record fields are accessed by using the access operator: "."

In order to assign a value to the record field, you need to use the following expression:

```
<RecordName>.<FieldName1>{.<NestedFieldName2>...} = x
```

Example

```
Rec1.X = 1; Rec1.Y=2
Rec1.P[1,1] = 0
Rec1.S1 = "W100"
for i=1 to 10 do begin
   RR[i].X = i
   RR[i].Y = 0
   RR[i].Z = 50
end
rec3.field3.nestedField2[1]=33
```

You can use the value stored in the field as follows:

```
x = <RecordName>.<FieldName1>{.<NestedFieldName2>...}
```

Example

```
x1=Rec1.X; y1=Rec1.Y
z2=RR[2].Z
```

Objects

Variables of this type represent a complex structure consisting of fields and methods. Objects describe the structure of a dynamic variable that appears only during program execution. Such a variable is an instance of an object.

Properties are variables of type REAL, INTEGER, STRING, RECORD, or array. Object properties are data elements that are copied into each instance of the object.

Method is a procedure or function associated with an object. Inside methods, only the properties of the object instance for which the method was called are accessible. A function differs from a procedure in that it returns a result. It can be used in assignment statements, mathematical and string expressions, in the print statement, in logical expressions. The result of the function's action in its description is located in the predefined variable `result`.

Field declarations are made after the object declaration, method descriptions are made after the field declarations.

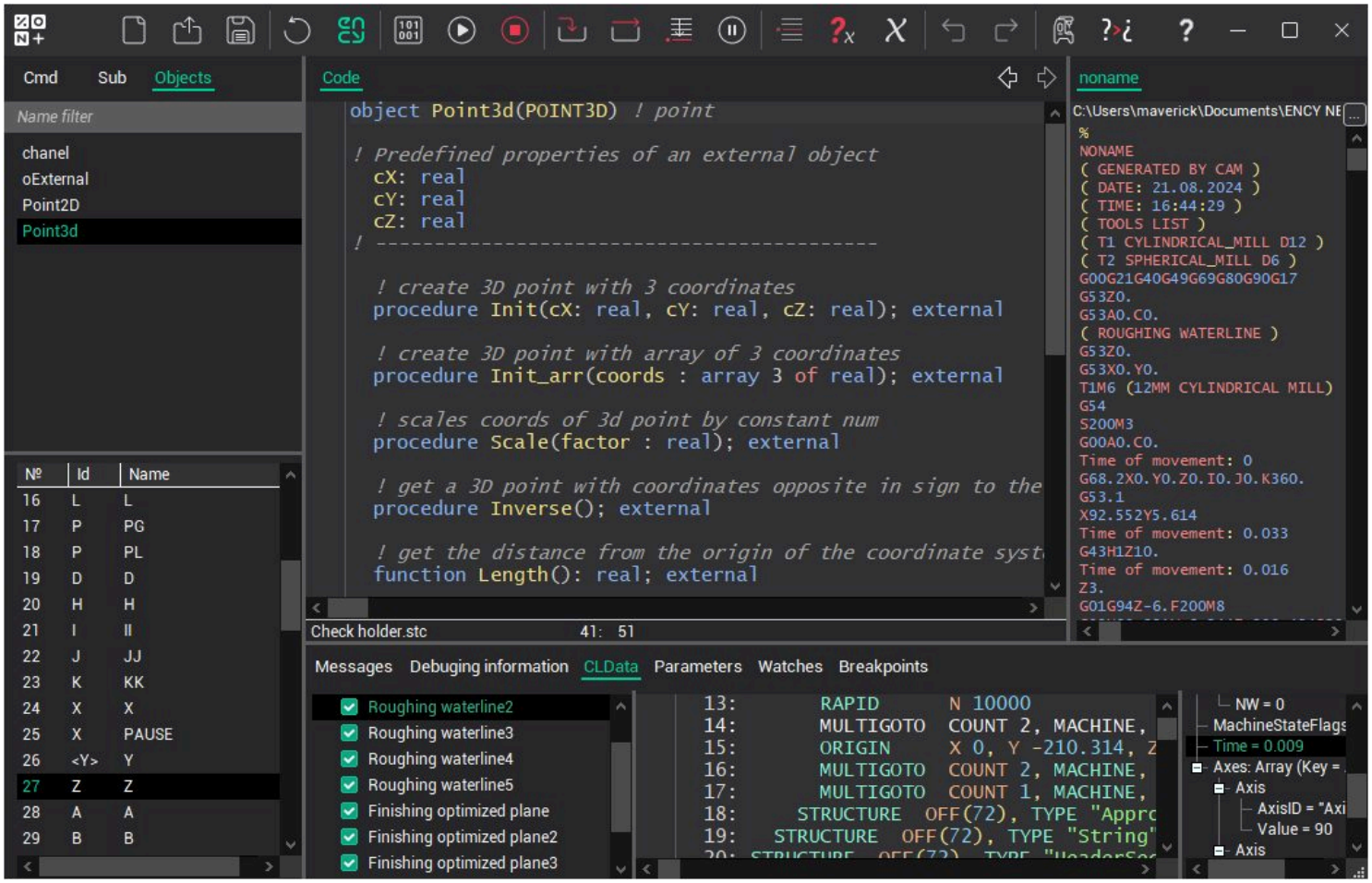
The predefined variable `self` is a hidden parameter for each method in the object. It allows the method to access the object. This is especially important when the method parameters are the same as the object variables intended to hold these passed values. Example:

```
object sample
  xx1, yy1, zz1: real;
procedure init(xx1, yy1, zz1: real)
begin
  self.xx1=xx1; self.yy1=yy1; self.zz1=zz1
end
```

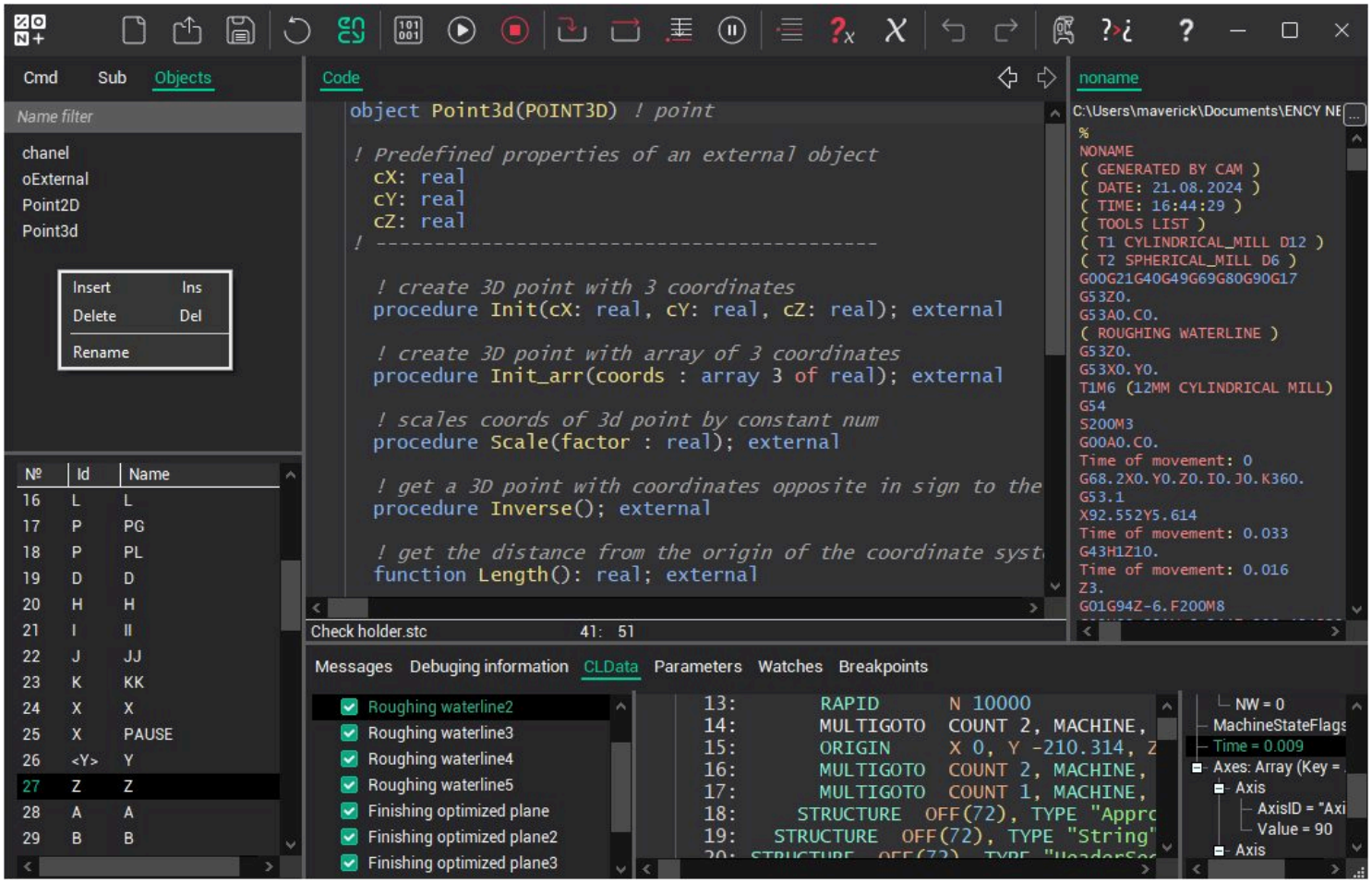
An example of sending of the object to subprogram:

```
object sample2
procedure proc
begin
  call subproc(self)
end
```

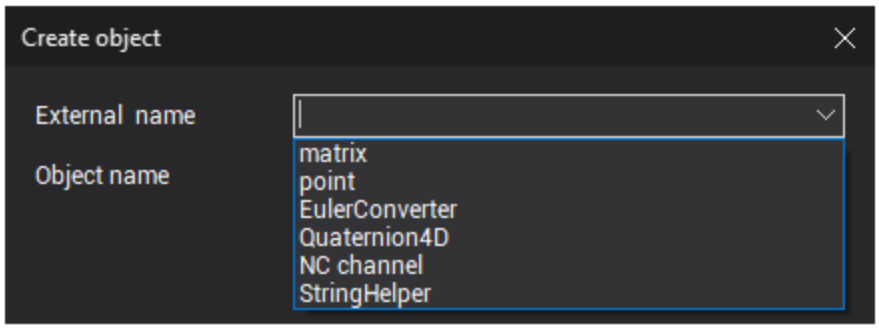
The object description ends with the command `end.`.



Creating a new object, deleting or renaming an existing object is done through the context menu in the object list.



When creating a new object, a window will appear:



Here External name is the name of the external library, the entire implementation of which is done in Delphi. Object name is the name under which the object will be accessible in the system.

Currently available:

- EulerConverter – spatial rotations using Euler angles.
- Matrix – three-dimensional matrix.
- Point3D – three-dimensional point.
- Quaternion4d – quaternion (4D vector).
- NCChannel – solves the problem of managing multiple channels simultaneously.
- StringHelper – basic functions for working with strings, dates and times.

Note: you can create heirs from the presented objects, to which you can add your parameters, but only after the existing parameters!

Declaration of an object instance is done in commands or subprograms similarly to arrays and structures.

Example of declaring and using an object instance:

```
pp: point2D
pp.xp=10
pp.yp=10
pp.MoveV(45, 10)
pp.TurnAndMove(45, 15, 25)
print pp.xp, "; ", pp.yp
```

Objects can be assigned entirely, like ordinary variables. Continuing the previous example:

```
pp2: point2d
pp2=pp
```

Also, like ordinary variables, objects can be parameters of procedures, functions, and subprograms:

```
DrawPoint(pp)
```

EulerConverter

Euler Converter – object of 3D rotation converter.

Parameters

- RotatingSequence: integer, options:
  - 1 – XYZ
  - 2 – XZY
  - 3 – YXZ
  - 4 – YZX
  - 5 – ZXY
  - 6 – ZYX
  - 7 – XYX
  - 8 – XZX
  - 9 – YXY
  - 10 – YZY
  - 11 – ZXZ
  - 12 – ZYZ
  - 13 – AB
  - 14 – AC
  - 15 – BA
  - 16 – BC
- MovableOrFixed: integer:
  - 1 – Rotating around movable axes
  - 0 – Rotating around fixed axes
- AngleInDegrees: integer
  - 1 – Degrees
  - 0 – Radians

Methods

- Init – create Euler Converter Object
- AnglesToMatrix – create translation matrix with 3 angles and 3D point
- MatrixToAngles – get 3 angles and point from translation matrix
- AnglesToAngles – translate 3 angles of input Convention to 3 angles of output convention
- AnglesToQuaternion – translate 3 angles of input convention to quaternion

Matrix

Three-dimensional matrix

Parameters

- Vx: record
  - cX, cY, cZ: real
- A: real
- Vy: record
  - cX, cY, cZ: real
- B: real
- Vz: record
  - cX, cY, cZ: real
- C: real
- Vt: record
  - cX, cY, cZ: real
- D: real

Methods

- CreateIdentity – create unitary matrix
- CreateBy4points – create matrix with four 3D points
- MakeRotMatrix – create rotational matrix with angle, angleType (1 for degrees or 0 for radians) and number of an axis (1 for "X", 2 for "Y", 3 for "Z")
- MakeShiftMatrix – create translation matrix with 3D point
- CreateAsAxisAngle – creates matrix as axis and angle
- CreateByPointAndNormal – creates matrix by point and normal
- CreateByVectorAndAngle – creates matrix by vector and angle
- Add – add another matrix to the original
- Subtract – subtract another matrix from the original
- Matr_X_Matr – multiply the original matrix by another
- GetDeterminant – get determinant of a matrix
- GetCoord – get coord by row and column of matrix
- Inverse – invert matrix
- Transpose – transpose matrix
- CreateBy3Points – create a basis with three 3D points
- GetLocalPoint – represent global 3D point in local coordinate system
- TransformPoint – represent local 3D point in global coordinate system
- GetLocalVector – represent global 3D vector in local coordinate system
- TransformVector – represent local 3D vector in global coordinate system
- GetLocalMatrix – represent global 3D matrix in local coordinate system

- TransformMatrix – represent local 3D matrix in global coordinate system
- ToString – creates string with numbers from the matrix
- Parse – creates 4x4 matrix from the string

NCChannel

The Channel object solves the problem of managing multiple channels simultaneously. Each channel manages its own isolated list of registers. The Activate procedure allows you to switch the currently active channel. The OutBlock, FormBlock, and Output procedures allow you to perform the appropriate functions for a channel without switching it to active mode.

Parameters

- FileName: string
- ChName: string – channel name.

Methods

- Init – object initialization.
- Activate – switching the channel to the current state.
- OutBlock – this statement forms the content of a block, corresponding to specified variable format, order, register format, and outputs the block into NC-program.
- FormBlock – this statement forms the block of NC-program according specified format, order and registers format. The block is formed in OutStr$ variable without the output in the file of NC-program.
- Output – string output
- IsActivCh – is active channel, 1 - yes, 0 - no.
- CloseFile – fixing the state of the current file.

Registers access methods:

- PutReg – put value to register.
- GetReg – get value from register.

Point3D

Parameters

- cX, cY, cZ: real – coordinates of point.

Methods

- Init – create 3D point with 3 coordinates
- Init_arr – create 3D point with array of 3 coordinates
- Scale – scales coords of 3d point by constant num
- Inverse – get a 3D point with coordinates opposite in sign to the original ones
- Length – get the distance from the origin of the coordinate system to a 3D point
- DistanceTo – get the distance from this 3D point to another 3D point
- Norm – get normalized 3D point
- Add – add another 3D point to the original
- Subtract – subtract another 3D point from the original
- VmV – get the scalar product of the original vector and another one
- VxV – get the vector product of the original vector and another one

Quaternion4d

Quaternion – the group of rotations about a fixed point in four-dimensional Euclidean space.

Parameters

- cX, cY, cZ, cW: real – coordinates of quaternion

Methods

- Init – create quaternion object
- q_mul_q – get dot product of quaternions
- q_x_q – get vector product of quaternions
- s_x_q – scale quaternion coordinates
- q_add_q – add quaternion
- q_sub_q – subtract quaternion

StringHelper

Methods

- Split – splitting a string into an array of substrings
- Join – concatenating an array of strings through a delimiter into one string
- Insert – inserting one string into another
- ReplaceStr – replacing a character or substring in a string
- Substr – taking a substring by starting element index and length
- CurrDate – returning the current date in standard format
- CurrTime – returning the current time in standard format
- FormatDateTime – returning the current time or date in custom format
- FormatReg – returns a string and a number formatted according to the register pattern

Operators

No content in this page. See child topics

The processing program start operator PROGRAM

Each processing program of the technological commands begins by the header, which consists the PROGRAM keyword and the program name, and terminates by the keyword END.

Format

```
PROGRAM ProgramName
  <statement 1>
  ...
  <statement N>
END
```

Description

The text of a program is written on the special problem-oriented language and can contain the mathematical expressions, functions, input/output operators, conditional operators, cycles, jump statement, calls of subroutines, the statements to form the NC-program blocks and the statements to work with the file of the technological commands.

Example

```
PROGRAM AbsMov
  FormBlock
  X = cld[1]
  Y = cld[2]
  if X <> LastX or y <> LastY then begin
    xs$ = Str(40*x)
    ys$ = str(40*y)
    if IsRapid> 0 then fs$ = "PU"
    else fs$ = "PD"
    output fs$ + xs$+","+ys$+";"
    LastX = X
    LAstY = Y
  end;
END
```

The statement of assignment

This statement is used to assign the value of an expression to the variable. The variables may change their values because of performing of this statement.

Format

```
<Numerical variable> = <Math expression>
```

or

```
<String variable> = <String expression>
```

Description

The keyword of this operator is the variable name. The symbol = follows after the name, and then follows the expression, the value of this expression will be assigned to variable. This operator defines that the expression interpretation result must be stored in the memory cell, identified by specified variable. The variable type must be coincident with the type, returned by the expression, i.e. the value of the math expression must be assigned to the numerical variable, and the value of the string expression – to the string variable.

Note: When the statement is performed, the value of right expression is evaluated first, and then, retrieved value will be assigned to the variable in the left part of statement. This rule allows using the same variable in the left and right parts of statement. In this case, the old value of variable participates in the evaluation of the right expression and, after this; the value of right expression will be assigned to the same variable.

Examples

```
! Usage of the statement statement
Zet = 41
Ddel = Mn / Cos(Beta) * Zet
Number = Number + 5
! The generation of string about the NC-machine
System$ = " 2C42 "
! The assignment of CNC-name to the System$
NAME$ = " TEST PROGRAM " + System$
```

The output statement PRINT

This statement is designed to write the results on the display (in the debug window) during the trial NC-program generation. If the run-time postprocessor (`Inp.exe`) generates the NC-program, it ignores this operator.

Format

```
PRINT <Math expression> | <String expression>
{, <Math expression> | <String expression>}
{ > | >> output file name}
```

Description

The keyword of this statement is the PRINT word. The list, which contains one or more math or string expressions, follows the keyword. The number of expressions is not limited in this statement, if there are more than one expressions in the statement, they must be separated by commas.

The value of expressions will be consecutively evaluated and printed in the Debuging information tab.

To save the output information on disk, you must redirect the output to a file. To do this, it is enough to specify the symbol > or >> after the last of the output expressions and the name of the output file.

The use of the symbols > and >> indicates the mode for outputting information to a file:

- symbol > - start output from the first record, having previously cleared the output file. If a file with the same name already exists, the old contents of the file are lost;
- symbol >> – add output parameters to the end of the file. If there is no file with that name, it will be created and the contents of the current output statement will be the first entry in the new file.

Examples

```
PRINT "Runs the subprogram ABSMOVE"
PRINT CLD[1]-XT, " increase by X"
PRINT AA, " ", CLD[1]


PRINT CLD[1] > "c:\my_folder\log.txt" ! Output to file
```

See also

[The input statement INPUT]

The input statement INPUT

This operator is intended to input user data from the keyboard or from the file during postprocessor program module execution.

The input from keyboard

The input from file

See also

The output statement PRINT

The input from file

Syntax

```
INPUT <variable> | <string variable> < | << "<file name>"
```

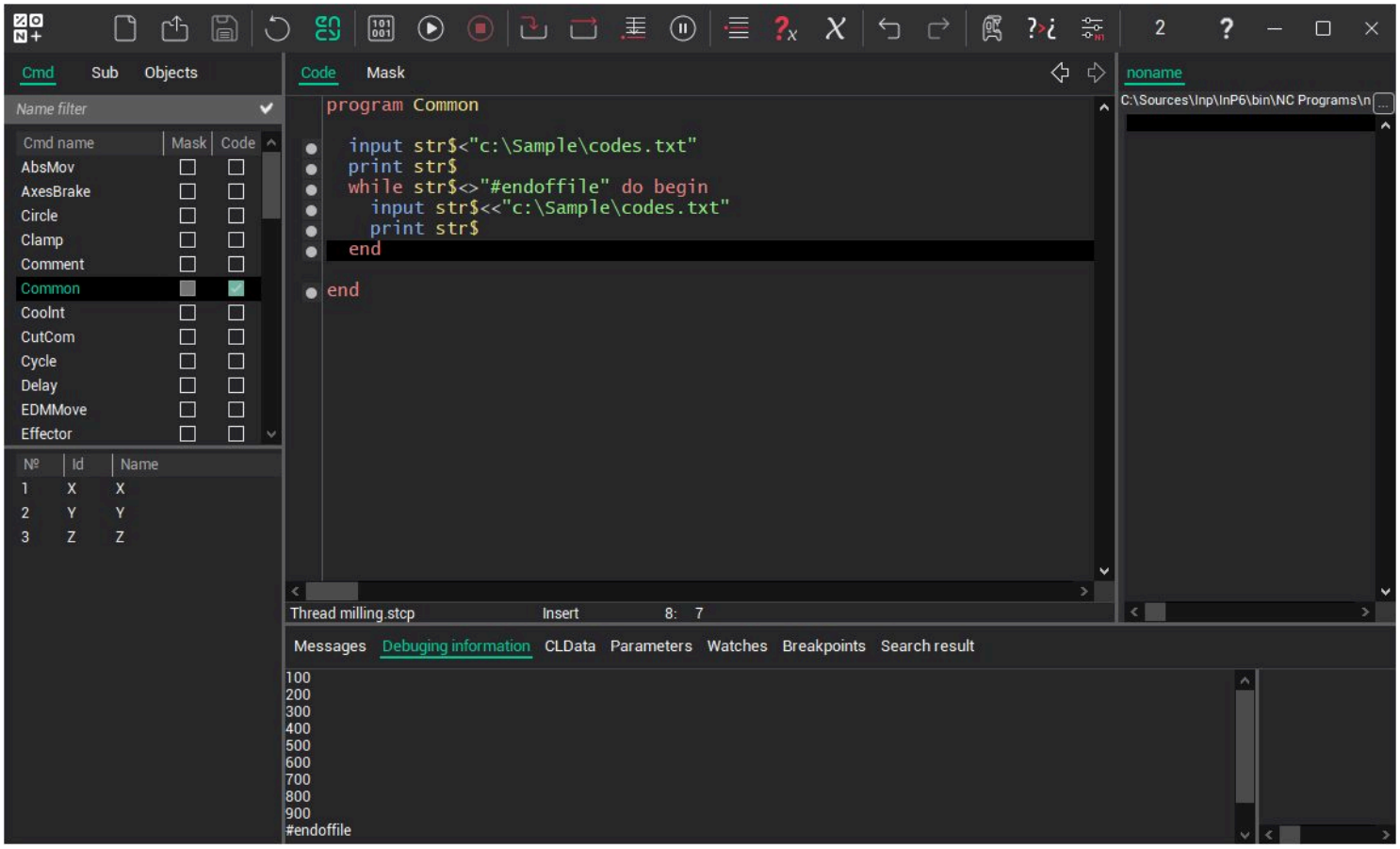
Description

For file-based disk input, redirect input by adding < or << symbols and the file name after parameters. In this case, when executing the operator, the quoted message (string literal) is displayed on the screen, and data input is performed from the specified file.

- symbol "<" is used to start inputting values into the current input operator's variables from the file's initial line.
- using the "<<" sign continues input operations from the line immediately after the previous input operation's end point in the file.

When the final line is reached, the "#endoffile" value will be obtained.

Example



The input from keyboard

This functionality is intended to input user data from the keyboard during postprocessor program module execution.

Syntax

```
INPUT <caption>, <variable> | <string variable>
{, <caption>, <variable> | <string variable> }
{,<caption>, checkbox, <variable>}
{,<caption>,array, <variable>}
caption = "literal string"
```

Description

This operator uses the keyword INPUT, followed by a list containing one or more numeric or string variables and string literals. The number of parameters in the list is unlimited; if there is more than one, they are separated by commas.

Executing the operator creates an input window consisting of string literals and input fields. If variables were previously initialized before calling the operator, these values will be entered into the corresponding fields of the window as default values. During input, the system performs type checking for variables. Only numeric values can be entered into a numeric variable, while any characters can be entered into a string variable.

After entering all values, you must close the window by clicking the OK button. This assigns the entered values to their respective variables.

To set dialog window title pass "title##caption" as caption. Note that each subsequent "title##caption" parameter will override the dialog title.

To add an image to the input window, use the string "pic=filename". The input window will expand to accommodate the image.

The checkbox keyword generates a check box element. Initially, the check box is marked if the variable holds a non-zero value. The returned variable value equals zero for an unchecked state and non-zero for a checked state.

Array data input requires the inclusion of the array keyword. The array type can be string, real, or integer.

This operator functions similarly to an assignment operator, with the key difference that variable values are not assigned from within the program but are requested for input during each execution. Using this

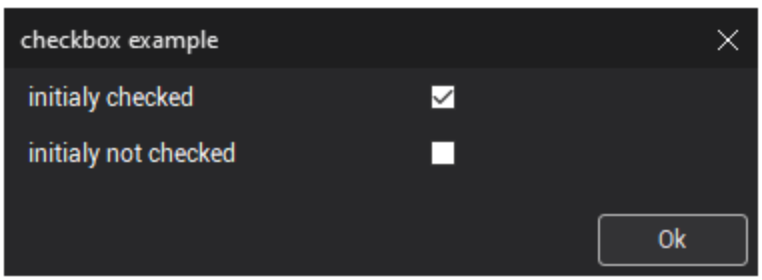
operator allows you to write a postprocessor program once and run it multiple times, entering different initial data each time to obtain new results depending on the input values.

Each edit field has two buttons to work with the Windows clipboard:

- – Copies the edit field text into clipboard (shortcut: [Ctrl+C]);
- – pastes the text from clipboard into the edit field (shortcut: [Ctrl+V]).

Examples

```
INPUT "Sample data input: ", Zt
MachineName$ = "Leader"
INPUT "Input the NC-machine name: ", MachineName$
wx = 10; wy = 10; wz = 5
INPUT "Workpiece dimensions",
"Along X axis", wx,
"Along Y axis", wy,
"Height by Z", wz
! Window title and name configuration via variables. notice the '""+' trick to make
PP use the variable caption and title:
rr = 2
title$ = "title"
caption$ = "caption"
input "" + title$ + "##" + caption$, rr
! Select variables from the list:
arr: array of real
arr[1] = 1.1
arr[2] = 2.2
input "caption", arr, rr
! Checkbox input example. notice the '""+' trick to make PP use the variable caption
title$ = "checkbox example"
caption1$ = "initialy checked"
caption2$ = "initialy not checked"
init_checked = 1
init_unchecked = 0
input "" + title$ + "##" + caption1$, checkbox, init_checked, "" + caption2$,
checkbox, init_unchecked
```



See also

[The output statement PRINT]

Conditional statement IF

This statement is designed to perform one of statements, included by it, in depending of some condition.

Format

```
IF <Conditional expression> then <Statement 1>
{else <Statement 2>}
```

Description

This operator allows changing the next steps of program dependently of some conditions.

The keyword of this statement is the IF word. Then follows the conditional expression, when it is true, the statement, following the THEN keyword, is performed. If the conditional expression is false, then the statement, following the keyword ELSE, is performed. ELSE – is unnecessary part of this statement. If it is absent and the value of conditional expression is false, then the performing of IF statement terminates and next statement of a program will be performed.

Conditional expression may be a single Simple conditional expression or a sequence of the Simple conditional expressions, linked by logical OR and AND operations, i.e.:

```
<Simple conditional expression> {{AND <Simple conditional expression>} |
{OR <Simple conditional expression>}}
```

Conditional expression is true, if all Simple conditional expressions, linked by AND operation are true or if at least one of Simple conditional expressions, linked by OR operation is true.

The Simple conditional expressions – is a two math or string expressions, linked by comparison operations, i.e.:

```
<Math expression> <Comparison operator> <Math expression>
```

or

```
<String expression> <Comparison operation> <String expression>
```

The expressions around the Comparison operation must be same type; otherwise, you will obtain the error message "Incompatible types" during the compilation.

The Comparison operation is one of following operations:

- = – the values of left and right operations are EQUAL;
- > – the value of left expression is GREATER than value of right expression;
- < – the value of left expression is LESS than value of right expression;
- # or <> – the values expressions are DIFFERENT;
- >= or => – the value of left expression is EQUAL OR GREATER than value of right expression;
- <= or =< – the value of left expression is EQUAL OR LESS than value of right expression.

It is necessary to pay attention to the comparison of string expressions. The comparison of two strings is performed character-by-character from left to right respecting their ASCII-codes. Keep in mind that capital letters are less than small letters.

If the lengths of the strings are different, but the longest string includes the shortest entirely, then the longest string is greater. The strings are equal, if they contain the same symbols and have the same length.

When the conditional statement is performed, the values of math and string expressions in the Simple conditional expressions are evaluated first, then the operations of comparison are performed, and then the AND and OR operations are performed without priority. After that, the THEN or ELSE statement will be performed, depending the result of entire Conditional expression.

Examples

```
! Sample 1.
! The example, which uses the conditional statement
IF kadr <224 THEN PRINT "This is a short program "
ELSE PRINT "You are the monster in programming"
! Sample 2.
! This example shows how to use embedded conditional statements
IF POS(" ",str$) = 2 THEN
b$ = "The first char in string is a letter"
ELSE
IF POS(" ",str$) <= 4 THEN
b$ = "First in string is a word, it is length is 4 symbols"
ELSE
b$ = "First in string is a word greater than 5 symbols length"
```

See also

Statement of the multiconditional execution CASE

Statement of the multiconditional execution CASE

The Case statement allows checking the value of the expression and depends of the result to execute different code.

Format

```
CASE <expression> OF
  <values list 1> : <statement 1>
  ...
  <values list N>: <statement N>
  ELSE <statement M>
END
```

Description

The expression must be numerical in this construction. Inadmissible to use a string expressions.

The values lists can contain one or more constants, divided by commas. The `:` symbol follows the values list. After that the statement is written that is executed if the expression is equal to one of the list. After the execution of this statement the execution of the CASE structure is finished and next statement is executed.

If the expression result is not defined in all lists then the statement after ELSE keyword is executed. The ELSE part is an optional. If ELSE absent and there is no corresponding value in the lists then nothing is executed.

Example

```
CASE i OF
  1,2,3,4,5: Str = "Less or equal 5"
  6,7,8,9: Str = "Greater 5"
  ELSE Str = "Error value"
END
```

See also

Conditional statement IF

JUMP statement

The operator is used to break the order of statements performing.

Format

```
JUMP <Label number>
...
<Label number>:
```

Description

After the JUMP keyword one of labels, which is defined in the current program, must be specified. The symbol : must not appear after the label number, unlike the label definition.

The operator, labeled by corresponding label, will be performed after the JUMP statement. it is necessary to define corresponding label, otherwise the error message will appear during compilation.

It is not recommended to usually use the JUMP statement in the program.

Example

```
AFF = 20 ! this statement is performed first
JUMP 2 ! jump to label 2
ABC = 1 ! This statement never be performed
2: ATR = 4 ! this statement is performed second
```

The cycle statement FOR

This statement repeats one or multiple operators a defined number of repetitions.

Syntax

```
FOR <numerical cycle control variable> = <math expression> TO <math expression>
{STEP <number> | <number variable>} DO <operator>
```

Description

This statement provides repetitive performing of some statement and simultaneous incrementing the cycle variable, until the value is greater than specified limit.

The keyword of this operator is the FOR word. Then follows the construction, which is similar to assignment:

```
<Numerical cycle control variable> = <Math expression>
```

In reality, they are much similar, because this construction assigns the initial value, defined by Math expression, to Numerical cycle control variable.

After the TO reserved word the math expression follows, which defines the upper limit. When the Numerical cycle control variable achieves this limit, the cycle statement terminates, and next program statement will be performed. Any math expression can be specified as an upper limit for cycle.

Unnecessary part of this statement, containing reserved word STEP and following number or numerical variable, defines the incrementing step for the cycle variable. As default, the step is 1. The expression can not be the step.

The statement, which must be performed cyclically, is specified after the DO keyword. All other cycle parameters serve for organization of cyclical performing of this statement.

When the loop body contains several operators, they should be placed between the keywords BEGIN and END.

Examples

```
! Example 1.
! Simple example of usage the cycle statement.
FOR as = 3 TO 10 DO PRINT as:0," ",as^2:0
! Example 2.
! Using embedded cycle statements
```

```
FOR i = 0 TO 0.9 STEP 0.5 DO
FOR j = -1 TO 0 STEP 0.2 DO
PRINT " i = ",i:1," j = ",j:1
```

See also

[The cycle statement REPEAT]

[The cycle statement WHILE]

[Statement BREAK]

[Statement CONTINUE]

Statement CONTINUE

Syntax

```
CONTINUE
```

Description

The CONTINUE statement interrupts the execution of the current iteration of a loop. In the case of a precondition loop (FOR or WHILE), control is transferred to the beginning of the loop, where the condition is checked (for the FOR loop, the new value of the loop variable is calculated before that). If the condition is true, the loop continues. In the case of a postcondition loop (REPEAT), control is transferred to the end of the loop, where the condition is checked. If the condition is false, the loop continues.

The use of the CONTINUE statement is only allowed within the body of a loop; in all other cases, an error message is issued.

See also

The cycle statement FOR

The cycle statement REPEAT

The cycle statement WHILE

Statement BREAK

Statement BREAK

Syntax

```
BREAK
```

Description

The BREAK statement is designed to terminate the execution of a loop. Control is transferred to the statement following the interrupted loop. In the case of nested loops, the execution of the loop in which the BREAK statement is located is terminated.

The use of the BREAK statement is only allowed within the body of a loop; in all other cases, an error message is issued.

See also

The cycle statement FOR

The cycle statement REPEAT

The cycle statement WHILE

Statement CONTINUE

The cycle statement REPEAT

The structure `REPEAT...UNTIL` is used for the repeating execution of the some operators named as cycle body, until some condition will be true.

Syntax

```
REPEAT
  <operators of the cycle body>
UNTIL <conditional expression>
```

Description

The structure works the following way. The operators of the cycle body is executed, after that is calculated the conditional expression. If the conditional expression returns false then the execution of the cycle body is repeated and after that, the conditional expression is calculated again. So the repeating is continues until the conditional expression returns true. After that, the cycle execution is finished and the next statement after structure `REPEAT...UNTIL` is executed.

Because the condition checking is performed after the execution of the cycle body then it is executed one time even if the condition is true in advance. On the other hand, the condition must return true one time else, the loop will be performed infinitely.

See also

The cycle statement FOR

The cycle statement WHILE

Statement BREAK

Statement CONTINUE

The cycle statement WHILE

The structure `WHILE...DO` is used for the repeating execution of the some operators named as cycle body while some condition is true.

Format

```
WHILE <conditional expression> DO <statement>
```

The structure works the following way. At the first, the conditional expression is calculated. If the result is true then the cycle body is performed and after that the conditional expression is calculated again. So the repeating is continues until the expression returns false. After that the cycle execution is finished and the next statement after structure `WHILE...DO` is executed.

Because the condition checking is performed before the execution of the cycle body then if the conditional expression is false in advance then the cycle body is never performed. It is the main difference between the structures `WHILE...DO` and REPEAT...UNTIL. On the other hand the condition must return true one time else the loop will be performed infinitely.

See also

The cycle statement FOR

The cycle statement REPEAT

Statement BREAK

Statement CONTINUE

Composite statement BEGIN...END

It is used to unite a group of statements to a one statement. It is the analogue of the parentheses

Format

```
BEGIN
<statement 1>
...
<statement N>
END
```

Description

The composite statement provides the execution of the included statements step by step. The auxiliary words BEGIN and END are analogue of the parentheses. The multiplicity level of the composite statements is unlimited. The composite statement can be used everywhere where a simple statement can be used.

Example

```
! Simple example of the composite statement
IF var < 3 THEN BEGIN
  ac=12; bb=16
END ELSE BEGIN
  ac=15; bb=60
END ! word END close the statement
```

Statement to call a subprogram CALL

The operator is intended to call the subprogram from the CLData command-processing program or other subprogram.

Format

```
CALL <program name> {(<the list of the parameters>)}
```

Description

The keyword of this operator is the CALL word. After this word it is necessary to define the program name – string expression or the literal string without the double quotation marks. The expression value is the name of the subprogram.

The list of the parameters a comma separated list of actual parameters: variables, literal constants, expressions and arrays. Types of actual parameters must match the types of formal parameters specified in the subprogram declaration. If no parameters are passed to the procedure then there must be no parenthesis.

The CALL jumps the execution to the defined subprogram and after the execution makes the return to the program, the call was raised from.

Example

```
X = 50; C = 0
CALL ConvertXYtoXC(X, 30, C)
CALL FillCycleParameters
```

See also

[The statement of the subprogram start SUB]

[The statement of the subprogram end SUBEND]

The statement of the subprogram start SUB

It is used for the declaration of the subprogram and the parameters list that is passed to the program then call.

Format

```
SUB <subprogram name> {(<the list of the parameters>)}
```

Description

The keyword of this operator is the SUB. Then follows the sub-program name – literal string without the double quotation marks and after that the optional parameters list in the parentheses.

The list of the parameters is the sequence of the numerical and string variables or arrays. If the parameters number is more the one then it is divided by commas.

The variables defined if the list will contain the values defined in the call statement when call. Therefore, these variables are declared in subprogram and can be used everywhere.

Example

```
sub GetProgramID(PrgID: Integer)
  ! Extracting numeric identifier from the string NC-program name
  i: Integer
  j: Integer
  k: Integer
  s$: String
  i = 1
  j = 0
  k = 0
  while i<=Len(NCName$) do begin
    s$ = Copy(NCName$, i, 1)
    case Ord(s$) of
      48, 49, 50, 51, 52, 53, 54, 55, 56, 57: begin
        if j<1 then j = i
        if (k<1) or (k=(i-1)) then
          k = i
      end
    end
    i = i + 1
  end
  if (j>0) and (k>0) then begin
    s$ = Copy(NCName$, j, k-j+1)
```

```
    PrgID = Num(s$)
  end
subend
```

See also

Statement to call a subprogram CALL

The statement of the subprogram end SUBEND

The statement of the subprogram end SUBEND

The statement is used for the declaration of the end of the subprogram.

Format

```
SUBEND
```

Description

The statement is declaring the end of subprogram. At the SUBEND execution the subprogram execution is closed, the values of all variables defined in the parameters list is assigned to the variables defined in the CALL statement and the execution of the parent subprogram is continued.

See also

Statement to call a subprogram CALL

The statement of the subprogram start SUB

The statement of the procedure start PROC

The operator is intended to declare the start and the parameters list of the subprogram in the CLDATA command processing procedure.

Format

```
PROC <procedure name> {(<list of formal parameters>)}
```

Description

After the PROC keyword, the procedure name is declared. The procedure name – identifier it is not coincide with the postprocessor generator keywords and variable names. This name is for the procedure indentification for calling, so the names of all procedures must be unique. After the procedure name follows the optional list of formal parameters. It is the sequence of the numerical and string variables and arrays. If the number of parameters is more then one then comma must divide it.

As distinct from SUB statement the variables defined in the list of formal parameters is not initialized i.e. it is necessary to assign some values to the variables before the including to the list.

The variables declared in the list of formal parameters will contain the values that are defined in the CALL statement. So these variables is defined in a program and can be used everywhere.

The PROC statement is looks like SUB statement. The main difference is that the procedure has the access to all variables and arrays of program of subprogram in the body of which it is located.

It is possible to place the declared procedure in any place of a program.

See also

The RETURN from a procedure statement

The RETURN from a procedure statement

It is used to declare the end of procedure.

Format

```
RETURN
```

Description

This statement is the last statement of a procedure. While the execution of the RETURN statement the back assignment of the formal parameters to the variables defined in the call statement is executed and start the execution of the next after the call statement.

Example

```
program TestProc
  NErr: Integer
  S: String
  proc PrintDebugInfo(NErr, S)
    case NErr of
      1: Print "Interpolation error: ", S
      2: Print "Approximation error: ", S
      else Print "Unknown error: ", S
    end
  return
  PrintDebugInfo(CLD[1], "parameter 1")
  PrintDebugInfo(CLD[2], "parameter 2")
  PrintDebugInfo(CLD[3], "parameter 3")
end
```

See also

[The statement of the procedure start PROC]

The block output statement OUTBLOCK

This statement forms the content of a block, corresponding to specified variable format, order, register format, and outputs the block into NC-program.

Format

```
OUTBLOCK {<FileName$>}
```

Description

The block of NC-program is formed according following algorithm:

The registers are checked from the first. If the previous and current values of register are different, then the register will be written in the block, and its current value will be assigned to its previous value; otherwise the register won't be written in the block.

When the register is written in the block, its identifier will be written first, then it's value, multiplied by the scale. The number will be written in the block corresponding to specified format. After that, the processing program will be called, if it is defined. The block is formed in the variable OutStr$, and will be written in the program as separate string after the formation is done.

Use optional `FileName$` parameter to specify the file where the system outputs the NC-block. If the parameter is omitted then the NC-block is output into the current file. If "*" is specified as a file name then the NC-block would be output into the main file, also the current file will be changed. If `FileName$` specifies the name of a file then corresponding tab will be created and the current file would be changed accordingly.

See also

Predefined functions

The block forming statement FORMBLOCK

Statement of direct output into the block OUTPUT

The block forming statement FORMBLOCK

This statement forms the block of NC-program according specified format, order and registers format. The block is formed in OutStr$ variable without the output in the file of NC-program.

Format

```
FORMBLOCK
```

Description

The formation process is similar to the OUTBLOCK statement, but the OutStr$ variable won't be written in the block of NC-program. It's possible, then, to work with this variable as a string variable and put it into NC-file by OUTPUT statement.

See also

Predefined functions

The block output statement OUTBLOCK

Statement of direct output into the block OUTPUT

Statement of direct output into the block OUTPUT

This operator outputs the content of a string variable by separate block of NC-program.

Format

```
OUTPUT {<FileName$>}, S$
```

Description

`S$` – the identifier of a string variable. The literal constant or string expression may be the parameter of this operator.

Use optional `FileName$` parameter to specify the file where the system outputs the NC-block. If the parameter is omitted then the NC-block is output into the current file. If "*" is specified as a file name then the NC-block would be output into the main file, also the current file will be changed. If `FileName$` specifies the name of a file then corresponding tab will be created and the current file would be changed accordingly.

Examples

```
! The output of the first symbol into NC-code
OutPut "%"
Name$ = " John "
F$ = "Lord"
! The output of string into NC-program
OutPut "Programmer "+Name$+" "+F$
! The string "Programmer John Lord" will be written into NC-program
! Sample of usage the FormBlock and Output usage instead of OutBlock
FormBlock
Output OutStr$
! Two these statements work as OutBlock
```

See also

[The block output statement OUTBLOCK]

[The block forming statement FORMBLOCK]

The replace statement of the substring in a string REPLACE

The statement is looking for the substring in a string and replace to the required.

Format

```
REPLACE (<string variable>, <looked string>, <string for replace>)
```

Description

This operator is looking for the looked string from the start of string variable. If looked string is found then it is replaced on the string for replace and the search is break. if the looked string is not found then replacement don't take the place.

Example

```
! Filling the source variable:
S$ = "GXX"
! Search and replace "XX" on "21":
REPLACE(S$, "XX", "21")
! The variable S$ has the "G21" value.
```

The statement to form the block by mask MASK

The Mask statement forms the block of NC-code by mask string. According to the rules and output the string to the NC-program.

Format

```
MASK (<mask string>) {, OutBlock | FormBlock {, Poll}}
```

Description

The statement masks form the NC code line according to the rules that is described in the mask string.

Options OutBlock and FormBlock are mutually exclusive.

Poll option corresponds to the Poll all registers switch. See Mask switches topic for details.

- OutBlock corresponds to Out Block switch;
- FormBlock starts OutStr$ forming mode similar to FORMBLOCK, without output to NC-text.

Example

```
! Form the NC code lines by mask
MASK(M[30]), OutBlock
MASK(% N[Off] ), OutBlock
MASK(G_INTERP[INTERP]X[CLD.X]Y[CLD.Y]Z[CLD.Z]), OutBlock, Poll
! Form NC-line in OutStr$ variable
MASK(G_INTERP[INTERP]X[CLD.X]Y[CLD.Y]Z[CLD.Z]), FormBlock, Poll
! OutStr$ holds formed NC-line
Output OutStr$
```

Change current NC-file operator

ChangeNCFile changes the current output NC-file. Thus following NC-code output operators which do not specify NC-file will write NC-blocks into the current NC-file.

Syntax

```
ChangeNCFile NewNCFileName$
```

Description

`NewNCFileName$` – expression that evaluates into string result.

Example

```
! Changing current NC-file sample
ChangeNCFile "File1.nc" ! changing the current NC-file
Output "NC block" ! this line will be output into the "File1.nc" file
```

See also

The block output statement OUTBLOCK

Statement of direct output into the block OUTPUT

External task call operator

Use ShellExec operator to execute a shell command or to call an external task.

Syntax

```
ShellExec CommandLine$
```

Description

`CommandLine$` – string expression.

Then executing this operator Postprocessor will run the operating system shell command interpreter (CMD.EXE) and pass it the `CommandLine$` as command (this operator is effectively executing "cmd /c CommandLine$"). Note that the console window is not displayed.

Example

```
! OS command execution
ShellExec "erase temp\*.*" ! Clear the temp folder
```

CustomData access operator CAM system

Use CustomData operator to access CustomData of a CAM system project.

Syntax

```
S$ = CustomData(ItemName$)
```

Description

`ItemName$` – expression that evaluates to string.

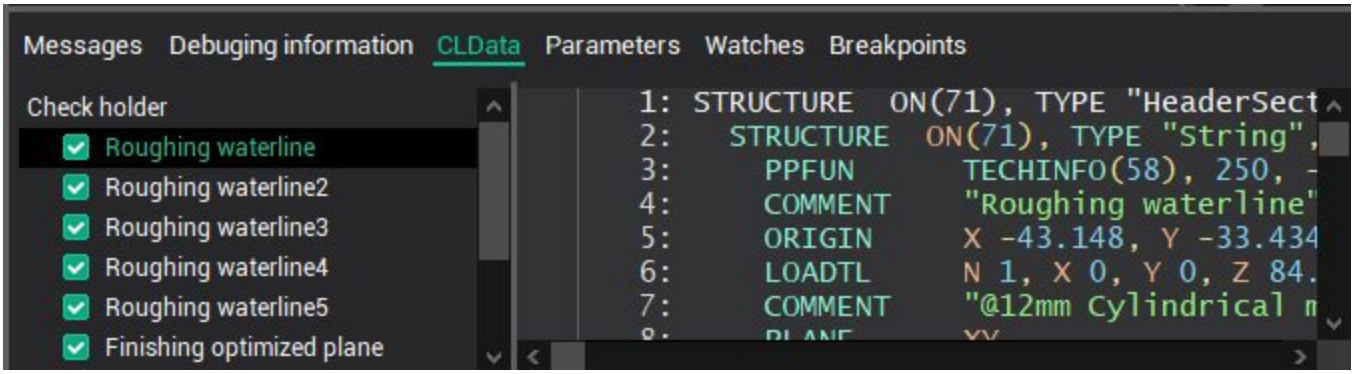
CustomData operator returns the value of custom data item named ItemName$ set by the user in the CAM system project.

Example

```
! Output user data into NC-program:
Output CustomData("UserData")
```

Operators of work with NC-subroutines

The NC-subroutine represents the sequence of technological commands issued in the form of a separate file of technological commands *.mcd, framed with special technological commands of the beginning of NC-subroutine PPFUN STARTSUB(50) and the end of NC-subroutine PPFUN ENDSUB (51).



As is shown in drawing in the list of files of technological commands a files with NC-subroutines are marked with grey color, and change of a condition of a tick near to a file name becomes impossible.

Unlike usual files *.mcd, technological commands in which start to be processed automatically if the corresponding file is marked by a tick near to the name, technological commands in NC-subroutines will be processed only by a call of the special operators beginning with a keyword NCSUB, for example, in the course of processing of technological command PPFUN CALLSUB(52).

Each NC-subroutine is identified by unique number which is defined in parameters of technological commands PPFUN STARTSUB(50), PPFUN ENDSUB(51) and PPFUN CALLSUB(52) the number 2 is accessible in processing procedures through predetermined file CLD. Thus, in procedure of processing expression `CLD[2]` will return number of the corresponding NC-subroutine. Also the NC-subroutine can be characterized by a text name. Initial value of a text name is define in the technological command a COMMENT which place before technological command PPFUN STARTSUB(50). In procedures of processing of technological commands the text name of the NC-subroutine can be read, and also is modified by the operator NCSUB.NAME(n), where n – unique number of the NC-subroutine.

In the course of translation of technological commands in the operating program it can be demanded to identify text of the beginning and the end of the concrete NC-subroutine (to define markers or numbers of frame). For these purposes, are intended a operators of definition of initial and final markers of the subroutine NCSUB.STARTLABEL(n) and NCSUB.ENDLABEL(n), where n – unique number of the NC-subroutine.

See also

The operator of a output of NC-subroutine `NCSUB.OUTPUT`

`NCSUB.OUTPUTALL` operator

The operator of definition of a name of NC-subroutine NCSUB.NAME

The operator of definition of an initial marker of NC-subroutine NCSUB.STARTLABEL

The operator of definition of a final marker of NC-subroutine NCSUB.ENDLABEL

Technique of work with NC-subroutines

The operator of a output of NC-subroutine NCSUB.OUTPUT

The operator transfers control to a file of technological commands *.mcd, which accordance the certain NC-subroutine.

Format

```
NCSUB.OUTPUT {(<NC-subroutine Number> {, <the Mode>}<, File name>)}
```

The operator is function, i.e. returns result. Possible values of returned result: 0 or 1.

Description

As a result of performance of the operator programs of processing of technological commands of the NC-subroutine specified by the identifier `NC-subroutine Number` consistently start to be caused. NC-subroutine number is usually transferred through the predetermined file of parameters of technological commands CLD with an index 2. NC-subroutine number is optional parameter. In case of absence of number consecutive translation of all available NC-subroutines will be made.

The optional parameter Mode defines, in what cases the result of translation of the NC-subroutine should be deduced in the text of the operating program, and can accept three values: 0, 1 and 2. At not specified parameter Mode is considered, that the mode is equal 1.

In a mode 0 a translation of technological commands change of values of registers and global variables is made, however the result of translation does not output in the text of the operating program. Thus, "idle pass" is carried out, allowing to simulate a program call in that case when the text of the subroutine should appear in other place of the operating program.

In a mode 1 (by default) translation of technological commands of the NC-subroutine is made also as well as in a mode 0, however the result of translation is output to the text of the operating program if the NC-subroutine still was never deduced. By the next calls of the operator in this mode "idle pass", as well as in a mode 0 is carried out. Thus, the result of translation of the NC-subroutine is output to the operating program only once.

By a call of the operator in a mode 2 a result of translation of the NC-subroutine is deduced in the operating program always, i.e. it is so much times, the operator how many is caused. Such mode allows to "unfold" NC-subroutines, in a case if system CNC does not support performance of subroutines.

The returned result is show whether the NC-subroutine is output in the operating program (NC). The result is equal 0 if not was output (has been carried out "idle pass"). The result is equal 1 if the NC-subroutine was deduced in NC. Accordingly 0 the returned result always is equal in a mode 0, in a mode 2 – is

always equal 1. One result is equal in a mode 1 only after the first call of the operator for each NC-subroutine.

Use optional string parameter File Name to specify the name of the file, into which the NC-code of the subprogram will be saved. If the parameter is omitted then the subprogram will be output into the main NC-program file. If the File Name doesn't specify any file path the file of NC-subprogram will be created in the same folder as the main program.

Examples.

For example, the basic file of technological commands contains following commands:

```
0: PARTNO "Lathe Roughing"
1: PPFUN TECHINFO(58), 250.000, 114.890, 78.750, 0.000, 140.690,
2: COMMENT "Lathe Roughing"
3: LOADTL N 0, X 0.000, Y 0.000, Z 0.000, D 0.000, M 0, K 1, L 0.000
4: SPINDL ON(71), NO 1000.000, K 0, MODE CSS(2), SPEED 600.000
5: FEDRAT M 0.050, K 3, MMPR(316)
6: GOTO.abs X 140.690, Y 104.550, Z 0.000
7: GOTO.abs X 140.690, Y 78.750, Z 0.000
8: GOTO.abs X 114.890, Y 78.750, Z 0.000
9: FEDRAT M 0.050, K 0, MMPR(316)
10: PPFUN CALLSUB(52), 1.000, 0.000, 0.000, 0.000, 0.000, 0.000
11: FEDRAT M 0.050, K 8, MMPR(316)
12: GOTO.abs X 114.890, Y 104.550, Z 0.000
13: GOTO.abs X 140.690, Y 104.550, Z 0.000
14: SPINDL OFF(72), NO 0.000, K 0
15: FINI
```

While the file of the NC-subroutine with number 1 has the following list of technological commands:

```
1: COMMENT "Geometry_1"
2: PPFUN STARTSUB(50), 1.000, 0.000, 0.000, 0.000, 0.000, 0.000
3: FEDRAT M 0.000, K 0, MMPR(316)
4: GOTO.abs X 114.890, Y 33.381, Z 0.000
5: CIRCLE XC 114.690, YC 33.380, ZC 0.000, R 0.200, XE 114.831, YE
6: GOTO.abs X 110.151, Y 38.201, Z 0.000
7: CIRCLE XC 110.010, YC 38.060, ZC 0.000, R 0.200, XE 110.010, YE
8: GOTO.abs X 62.230, Y 38.260, Z 0.000
9: GOTO.abs X 41.460, Y 38.257, Z 0.000
10: PPFUN ENDSUB(51), 1.000, 0.000, 0.000, 0.000, 0.000, 0.000
```

Let's admit, that the program of processing of the tenth command (PPFUN CALLSUB(52)) the basic file looks so:

```
program PPFun
  if cld[1]=52 then begin! CallSub
    NCSub.Output(CLD[2])
  end
end
```

In that case serial processing of technological commands of the basic file since a line 0 till the line 9 will be made from the very beginning. Having reached a line 10, at processing of operator `NCSub.Output(CLD[2])`, management will pass to a file with the NC-subroutine number 1 since the second parameter of a technological command is equal 1.000. Thus, technological commands of the NC-subroutine since a line 1 till the line 10 will start to be processed consistently. After processing of last command (PPFUN ENDSUB(51)) management will return to the basic file, and all remained commands (lines 11 – 15) will be processed.

Other examples of a call of the operator are resulted is shown below:

```
! Carries out translation of all available NC-subroutines
NCSub.Output
! Carries out NC-subroutine translation with number 3 in a mode 0
NCSub.Output(3, 0)
! Carries out NC-subroutine translation with number taken from the
! second parameter from a current technological command in a mode 2
NCSub.Output(CLD[2], 2)
if NCSub.Output(CLD[2]) = 0 then ! Carries out translation
begin ! NC-subroutines in a mode 1.
Output "CALL "+ ! Deduces a line "CALL ", if already
NCSub.StartLabel(CLD[2]) ! there was made a output in NC
end
! Process NC-subprogram number 7 in mode 1
! and output into Sub7.txt file
NCSub.Output(7, "Sub7.txt")
! Process NC-Subprogram number 7 in mode 2
! and output into Sub7.txt file
NCSub.Output(7, 2, "Sub7.txt")
```

See also

Operators of work with NC-subroutines

NCSUB.OUTPUTALL operator

The operator of definition of an initial marker of NC-subroutine NCSUB.STARTLABEL

NCSUB.OUTPUTALL operator

Use `NCSUB.OUTPUTALL` to output all NC-subprograms. Like the NCSUB.OUTPUT operator when called without parameters the `NCSUB.OUTPUTALL` operator translates all NC-subprogram CLData files, but unlike NCSUB.OUTPUT, `NCSUB.OUTPUTALL` can output all NC-subprograms into separate files.

Syntax

```
NCSUB.OUTPUTALL(<Mode>)
```

Mode – execution mode:

- 0 – works exactly like NCSUB.OUTPUT without parameters;
- 1 – operator will save NC-subprogram texts in files named NCSUB.NAME for each subprogram respectively.

The files extension is defined by the postprocessor properties. NC-subprogram files will be saved into the folder of the main NC-program.

See also

Operators of work with NC-subroutines

The operator of a output of NC-subroutine `NCSUB.OUTPUT`

The operator of definition of a name of NC-subroutine `NCSUB.NAME`

The operator of definition of an initial marker of NC-subroutine NCSUB.STARTLABEL

The operator returns a text value of an initial marker of the NC-subroutine, and also allows to define text value of an initial marker of the NC-subroutine any text value.

Format

```
S$ = NCSUB.STARTLABEL(<NC-subroutine Number>)
```

or

```
NCSUB.STARTLABEL(<NC-subroutine Number>) = S$
```

Description

- `NC-subroutine Number` — the unique identifier of the NC-subroutine.
- `S$` — any text variable. In the second case also can be set the text constant or any expression which is returning a text. The operator can be used by a similarly as text variable in expressions and functions processing strings.

The operator it is possible to use for definition of an initial markers of the NC-subroutine for a output in the operating program of the lines identifying the beginning of the subroutine. As a marker it is possible to set and a frame number.

Usually at the moment of the beginning of translation of technological commands in the operating program (NC) the location of the concrete subroutine in NC is not known. During this time the initial marker of the NC-subroutine is established in value by default SLabelNxxx, where xxx — NC-subroutine number. In the processing of NC-subroutine realization, i.e. its output in NC, usually is possible define an initial marker. Therefore in procedure of processing of a technological command of the beginning of subroutine PPFUN STARTSUB(50) use the operator of definition of an initial marker for installation of an initial marker of the subroutine:

```
NCSub.StartLabel(CLD[2]) = S$ ! Assignment of an initial marker
```

- `S$` — a text variable or any expression returning a text, corresponding to a marker or number of a frame of the subroutine.
- `CLD[2]` — NC-subroutine number transferred through the predetermined file of parameters of technological command CLD.

Note: At the included automatic numbering of frames of the operating program before the beginning of performance of procedure of processing of a technological command of the beginning of subroutine PPFUN STARTSUB(50) initial marker current value of the register of number of a frame automatically is appropriated. If it suits, it is possible not to use the operator for assignment of value of an initial marker.

Before a subroutine call usually is required to define, where the given subroutine is located. Therefore in procedure of processing of a technological command of a call of subroutine PPFUN CALLSUB(52) use the operator of definition of an initial marker of the subroutine

```
S$ = NCSub.StartLabel(CLD[2])! Reception of an initial marker
```

And then deduce the marker defined thus in the text of the operating program in the format necessary.

Note: If performance of procedure of processing of a technological command of a call of subroutine PPFUN CALLSUB(52) is made earlier, than there was an assignment of value of an initial marker (before the subroutine is deduced in NC) the operator of definition of an initial marker will return value by default a kind SLabelNxxx. After the termination of translation of the operating program if the subroutine has been deduced in CNC, and to a marker the new value differing from value by default has been appropriated, replacement of all values by default in text NC on again appropriated marker is made.

In this case the transformations which are carried out over line, a marker containing value (value by default a kind SLabelNxxx), should not break integrity of expression by default, differently the result of translation NC can appear incorrect. In such cases of transformation over line of a marker it is necessary to make not at reception of value of a marker, and before assignment of value of a marker in procedure of processing of a technological command of the beginning of subroutine PPFUN STARTSUB(50).

See also

Operators of work with NC-subroutines

The operator of definition of a final marker of NC-subroutine NCSUB.ENDLABEL

The operator of definition of a final marker of NC-subroutine NCSUB.ENDLABEL

Returns text value of a final marker of the NC-subroutine, and also allows to appropriate to text value of a final marker of the NC-subroutine any text value.

Format

```
S$ = NCSUB.ENDLABEL(
<NC-subroutine Number>
)
```

or

```
NCSUB.ENDLABEL(<NC-subroutine Number>) = S$
```

Description

- `NC-subroutine Number` — the unique identifier of the NC-subroutine.
- `S$` — any text variable. In the second case the text constant or any expression which returns text size also can be set.

The operator can be used as text variable in expressions and functions processing texts.

The operator is used for definition of a final marker of the NC-subroutine for a output in the operating program of the texts identifying the end of the subroutine. As a marker it is possible to indicate frame number.

Usually at the moment of the beginning of translation of technological commands in the operating program (NC) the location of the concrete subroutine in NC is not known. During this time the final marker of the NC-subroutine is established in value by default ELabelNxxx, where xxx — NC-subroutine number. In the course of NC-subroutine realization, i.e. its output in NC, usually is possible to define a final marker. Therefore in procedure of processing of a technological command of the end of subroutine PPFUN ENDSUB(51) use the operator of definition of a final marker for installation of a final marker of the subroutine:

```
NCSub.EndLabel(CLD[2]) = S$ ! Assignment of a final marker
```

- `S$` — a text variable or any expression returning a text, corresponding to a marker or number of a frame of the subroutine.

- `CLD[2]` — NC-subroutine number transferred through the predetermined file of parameters of technological command CLD.

Note: At the included automatic numbering of frames of the operating program before the beginning of performance of procedure of processing of a technological command of the end of subroutine PPFUN ENDSUB(51) final marker current value of the register of number of a frame automatically is appropriated. If it suits, it is possible not to use the operator for assignment of value of a final marker.

Before a subroutine call usually is define, where the given subroutine is located. Therefore in procedure of processing of a technological command of a call of subroutine PPFUN CALLSUB(52) use the operator of definition of a final marker of the subroutine

```
S$ = NCSub.EndLabel(CLD[2]) ! Reception of a final marker
```

Then output the marker defined thus in the text of the operating program in the format necessary.

Note: If performance of procedure of processing of a technological command of a call of subroutine PPFUN CALLSUB(52) is made earlier, than there was an assignment of value of a final marker (before the subroutine is deduced in NC) the operator of definition of a final marker will return value by default a kind ELabelNxxx. After the translation of the operating program terminated if the subroutine has been deduced in NC, and a marker the new value differing from value by default, will be made replacement of all values by default in text NC on again appropriated marker is made.

In this case the transformations which are carried out in line a marker of containing value (value by default a kind ELabelNxxx), should not break integrity of expression by default, otherwise the result of translation NC can appear incorrect. In such cases of transformation over line of a marker it is necessary to make not when get value of a marker, before assignment of value of a marker in procedure of processing of a technological command of the end of subroutine PPFUN ENDSUB(51).

See also

Operators of work with NC-subroutines

The operator of definition of an initial marker of NC-subroutine `NCSUB.STARTLABEL`

The operator of definition of a name of NC-subroutine NCSUB.NAME

Returns a text name of the NC-subroutine, and also allows to appropriate to a text name of the NC-subroutine any text value.

Format

```
S$ = NCSUB.NAME(<NC-subroutine Number>)
```

or

```
NCSUB.NAME(<NC-subroutine Number>) = S$
```

Description

- `NC-subroutine number` — the unique identifier of the NC-subroutine.
- `S$` — any text variable. In the second case also can be set the text constant or any expression which is returning a text.

The operator can be used by a similarly as text variable in expressions and functions processing strings.

For output in the operating program of the text identifier of the subroutine in the necessary format it is necessary to use the operator of definition of a name of the NC-subroutine.

Note: Before the first assignment of a name of the NC-subroutine by operator `NCSUB.NAME`, the name already matters by default which is taken from a text of the comment (technological command a COMMENT), located before a technological command of the beginning of NC-subroutine PPFUN STARTSUB(50) corresponding NC-subroutines.

Example

```
St$: text
n1: Integer
St$ = "Example"
n1 = 3
! Assignment of a name of value "Example3":
NCSub.Name(n1) = St$ + Str(n1)
! A output in the operating program of a text "oExample3":
Output "o"+NCSub.Name(n1)
```

See also

Operators of work with NC-subroutines

Technique of work with NC-subroutines

Existing systems CNC accept various ways of registration of subroutines in the operating program (NC), however by criterion of an order of the text of subroutines concerning lines of the basic program it is possible to allocate some typical cases:

- Subroutines are deduced in NC to a output of the basic program;
- Subroutines are deduced in NC after a output of the basic program;
- Subroutines are deduced in NC in a place of the first call;
- Subroutines are not supported, and the subroutine text is deduced by each call.

Consider how it is possible to realize each of the specified cases.

Subroutines are deduced in NC to a output of the basic program

The given format of a output assumes, that in the operating program all available subroutines is deduced at the beginning, and the basic program containing lines of a call of subroutines is inserted after.

To carry out a output of subroutines at the beginning, it is necessary to insert a corresponding code into procedure of processing of technological command PARTNO which is carried out before all. The procedure kind can be following.

```
program PartNo
  ! Initialization of variables
  ! A output of the lines identifying
  ! the beginning of section of subroutines
  NCSub.Output ! A output of all available subroutines
  ! A output of the lines identifying
  ! the end of section of subroutines and
  ! the beginning of section of the basic program
  ! Initialization of variables
end
```

For the purpose of formation of lines of the operating program identifying the beginning of the NC-subroutine, the end of the NC-subroutine and a NC-subroutine call, is necessary write procedure-output agent of technological commands PPFUN STARTSUB(50), PPFUN ENDSUB(51) and PPFUN CALLSUB(52) as shown below.

```
program PPFun
  if CLD[1]=50 then begin ! Command StartSub
    ! A output of the line identifying the beginning
    ! subroutines (here "NameSubroutine")
    Output "o"+NCSub.Name(CLD[2])
```

```
    ! Assignment to an initial marker current
    ! value of the register of number of a frame
    NCSub.StartLabel(CLD[2]) = Str(BlockN)
  end
  else if CLD[1]=51 then begin ! Command EndSub
    ! A output of the line identifying the end subroutines
    ! (here "Nxx M99", where Nxx - frame number)
    FormBlock
    Output OutStr $ + "M99"
    ! Assignment to a final marker values
    ! from the register of number of a frame
    NCSub.EndLabel(CLD[2]) = Str(BlockN)
  end
  else if CLD[1]=52 then begin ! Command CallSub
    ! A output of a line of a call of the subroutine of a kind
    ! "Nxx M98 Nbb, Nee", where
    ! Nxx - flowing a frame,
    ! Nbb - a beginning frame subroutine,
    ! Nee - a frame of the end of the subroutine
    FormBlock
    Output OutStr$ + "M98" +
    NCSub.StartLabel(CLD[2]) + "," +
    NCSub.EndLabel(CLD[2])
    ! Single run of the subroutine
    NCSub.Output(CLD[2], 0)
  end
end
```

In the given example the initial and final markers current values are appropriated from the register of number of a frame, however at the included automatic numbering, similar actions are made automatically and the specified code is optional. It is shown only for an example.

In the operator of a output of NC-subroutine NCSUB.OUTPUT the mode 0 is specified, however, since all subroutines are already deduced in the operating program earlier, it is possible not to determine the given parameter, that will be signify a output in a mode 1 ("only for the first time").

Subroutines are deduced in NC after a output of the basic program

Is supposed, that the text of the basic operating program together with lines of a call of NC-subroutines is placed before a code of subroutines. In this case the operator of a output of NC-subroutines is located in procedure of processing of a technological command which is processed by last – FINI, a appearance of this procedure is shown below:

```
program Fini
  ! A output of the lines identifying the end of section of
```

```
  ! the basic program and the beginning of section of subroutines
  ! Initialization of variables
  NCSub.Output! A output of all available subroutines
  ! A output of the lines identifying the end of section of subroutines
end
```

The example of the output procedure for technological commands of the beginning, the end and a call of subroutines (PPFUN STARTSUB(50), PPFUN ENDSUB(51) and PPFUN CALLSUB(52)) in comparison with the previous case remains invariable. However it is necessary to consider, that the values returned by functions of work with markers (NCSUB.STARTLABEL and NCSUB.ENDLABEL), till the moment of the termination of translation are not those markers which are appropriated by it, and have values by default a kind SLabelNxxx and ELabelNxxx, where xxx – NC-subroutine number, because real values of markers at the moment of inquiry are not defined. Real values of markers are substituted only after the termination of translation of the operating program as a whole.

Subroutines are deduced in NC in a place of the first call

In this case the subroutine is at the same time a part of the basic program, the reference to which can be made repeatedly, define for example, initial and final frames. Thus, the subroutine will be deduced by the first call as a part of the basic code, and at the subsequent – as a call of an existing code. All code which is responsible for the NC-subroutines, is located in procedure of processing of technological commands of the beginning, the end and a call of subroutines (PPFUN STARTSUB(50), PPFUN ENDSUB(51) and PPFUN CALLSUB(52)).

```
program PPFun
  if CLD[1]=50 then begin ! Command StartSub
    ! Assignment to an initial marker current value of
    ! the register of number of a frame
    NCSub.StartLabel(CLD[2]) = Str(BlockN)
  end
  else if CLD[1]=51 then begin ! Command EndSub
    ! Assignment to a final marker values from
    ! the register of number of a frame
    NCSub.EndLabel(CLD[2]) = Str(BlockN)
  end
  else if CLD[1]=52 then begin ! Command CallSub
    ! If an output of subroutine already was
    if NCSub.Output(CLD[2])=0 then begin
      ! Output of line of call of the subroutine of kind
      ! "Nxx M98 Nbb, Nee", where
      ! Nxx - flowing a frame
      ! Nbb - a beginning frame subroutine,
      ! Nee - a frame of the end of the subroutine
      FormBlock
```

```
      Output OutStr$ + "M98" +
      NCSub.StartLabel(CLD[2]) +
      ","+NCSub.EndLabel(CLD[2])
    end
  end
end
```

In the resulted example the first call of operator `[NCSUB.OUTPUT(CLD[2])](operator-of-a-output-of-nc-subroutine.md)` for each subroutine will carry out a NC-subroutine output in the operating program and will return value 1. By each subsequent call, "idle pass" will be carried out and value 0 will be returned. Then in the operating program, the line of a call of the corresponding subroutine will be added.

In the given example the initial and final markers current values are appropriated from the register of number of a frame, however at the included automatic numbering, similar actions are made automatically and the specified code is optional.

Subroutines are not supported

When system CNC does not support use of subroutines, and the sequence of technological commands contains NC-subroutines and their call it is possible to carry out "unfolding" of NC-subroutines, i.e. in all places of a call of the subroutine it is necessary to insert the subroutine instead of a line of a call. For this purpose it is necessary to insert a code of following kind into procedure of processing of a technological command of a call of subroutine PPFUN CALLSUB:

```
program PPFun
  if CLD[1]=52 then begin ! Command CallSub
    NCSub.Output(CLD[2], 2) ! A subroutine output in any case
  end ! (A mode 2)
end
```

Processing of commands of the beginning and the end of subroutines in this case is not required.

See also

Operators of work with NC-subroutines

The operator of a output of NC-subroutine `NCSUB.OUTPUT`

The operator of definition of a name of NC-subroutine `NCSUB.NAME`

The operator of definition of an initial marker of NC-subroutine `NCSUB.STARTLABEL`

The operator of definition of a final marker of NC-subroutine `NCSUB.ENDLABEL`

ReplNCStr operator to replace strings at NC-file

ReplNCStr operator replaces all From$ strings to To$ strings in the File$ file (or in all files, if the File$ is not specified or it is an empty string "").

Format

```
ReplNCStr(<From$>, <To$>{, <File$>})
```

Description

`From$`, `To$`, `File$` — string expressions.

Example

```
! Replace all "placeholder" strings to "world!" in all NC-files
Output "hello, placeholder" ! NC-file: hello, placeholder
ReplNCStr("placeholder", "world!", "") ! NC-file: hello, world!
! Replace one string of NC-file on two strings:
! Hello -> Hello,
! World!
Output "Hello"
ReplNCStr("Hello", "Hello," + Chr(13) + Chr(10) + "World!")
```

AI-assisted postprocessor development

This guide explains how to develop and maintain postprocessors with the help of artificial intelligence (AI): how to get the tools running, what to ask the assistant for, and how to check the result. It applies to both postprocessing subsystems — SPPX postprocessors and .NET postprocessors — because both read the same input data, CLData.

The toolset is in beta: command names and panels may still change.

What the work looks like

You write in the chat, in your own words:

> On this machine the coolant does not come on after a tool change. Find out why and fix it.

Then you watch. The assistant opens the test project, finds the command that should have produced that block, finds the handler that processes it, reads the code, makes a small change, compiles the postprocessor, runs it on the same project, and shows you the NC program it got and how it differs from the previous one.

You read the result and answer the same way — "right, but M08 has to come before the spindle starts" — and it goes round again.

That is the whole loop. Nothing to memorise, no commands to type, and you do not have to write the code yourself, although you can look at every change and edit it by hand whenever you prefer. What cannot be handed over is the judgement: whether the program is correct for your machine and safe on it is decided by you. Everything in this guide about checking the result exists for that reason.

What it costs

The tools and the knowledge are part of the product: the servers, the extensions, this documentation and the skills. The model that answers you is not — an AI assistant is a service of whoever provides the model, and it is paid for by the volume of text it processes.

In CAM Agent you can start straight away: it comes with an introductory allowance, enough to get a feel for what this can do. Its size and conditions may change over time. When it is used up, connect your own account with any of the publicly available AI providers; from then on the work goes on that provider's terms and at its prices, while the CAM system keeps supplying the tools and the knowledge.

Through a chat extension in Visual Studio Code, your provider account is what makes it work from the very first request. There is no introductory allowance in this case — the model is billed by the provider you connected, on the terms you agreed with it.

If the work must stay inside your own network, the model can live there too. Current tooling makes it practical to run one on your own hardware, and any client that can be pointed at a local model will use it. Nothing else in this guide changes: the servers, the extensions and the postprocessor itself are local anyway, so no project data has to leave the network. What you give up is the online half of the knowledge server — searching the published documentation goes through an online service, and in an isolated network it is not available, although skills stay searchable once their cache has been refreshed. The documentation in your workspace works regardless, as do your own notes; that is one more reason to keep a local copy even when everything else is connected.

Two ways to work

	CAM Agent	Visual Studio Code
Where you work	The agent chat inside the CAM system	VS Code with an AI chat extension
What you see	The task, the plan, the result; optionally the postprocessor IDE	The code, the diffs, the CLData, the generated NC program
Best for	Getting a result without opening the code	Developing, debugging and version-controlling a postprocessor
SPPX	Full cycle: read, edit, compile, run	Full cycle, with the code visible in the editor
.NET	Task-oriented use; the agent works with the project and the runner	The natural choice: C# language service, build, debugging

The two are not exclusive. Asking the agent in the CAM system to investigate the data and then finishing the code in VS Code is a normal way to work.

Read in this order

1. How the system works — the parts you will be dealing with, and which one does what.
2. Set up the tools — download the bundle and let the assistant do the setup.
3. Organize your workspace — the folder that gives the assistant a memory.
4. Develop with CAM Agent or Develop in Visual Studio Code — pick your entry point.
5. What the extensions give you — the panels and commands you will be using, and where they are.
6. SPPX workflow or .NET workflow — the development cycle for your postprocessor type.
7. Review and verify — what to check before the NC program is used.

8. Troubleshooting and Advanced setup — when something does not work, and the full setup procedure.

Skills for the assistant

A skill is a short instruction in plain language: what to do, in what order, what to check, and when the work counts as finished. There is no code in it — it is a text file the assistant reads and follows, much like a sensible briefing given to a new colleague. Anyone who can explain a way of working in words can write or correct one.

The skills that come with this guide cover setting up the workspace, inspecting the input data, the SPPX and .NET development cycles, and verifying a generated NC program.

They are published with this guide, in the `skills` folder of this documentation module — one folder each, with a `SKILL.md` file inside. The set grows as the toolset does; take all of it and let the assistant pick what fits the task. Loading them is part of the setup — see Advanced setup.

You can add your own alongside them: they are plain Markdown, and the practices your team accumulates belong in the same form — see Organize your workspace.

What stays with you

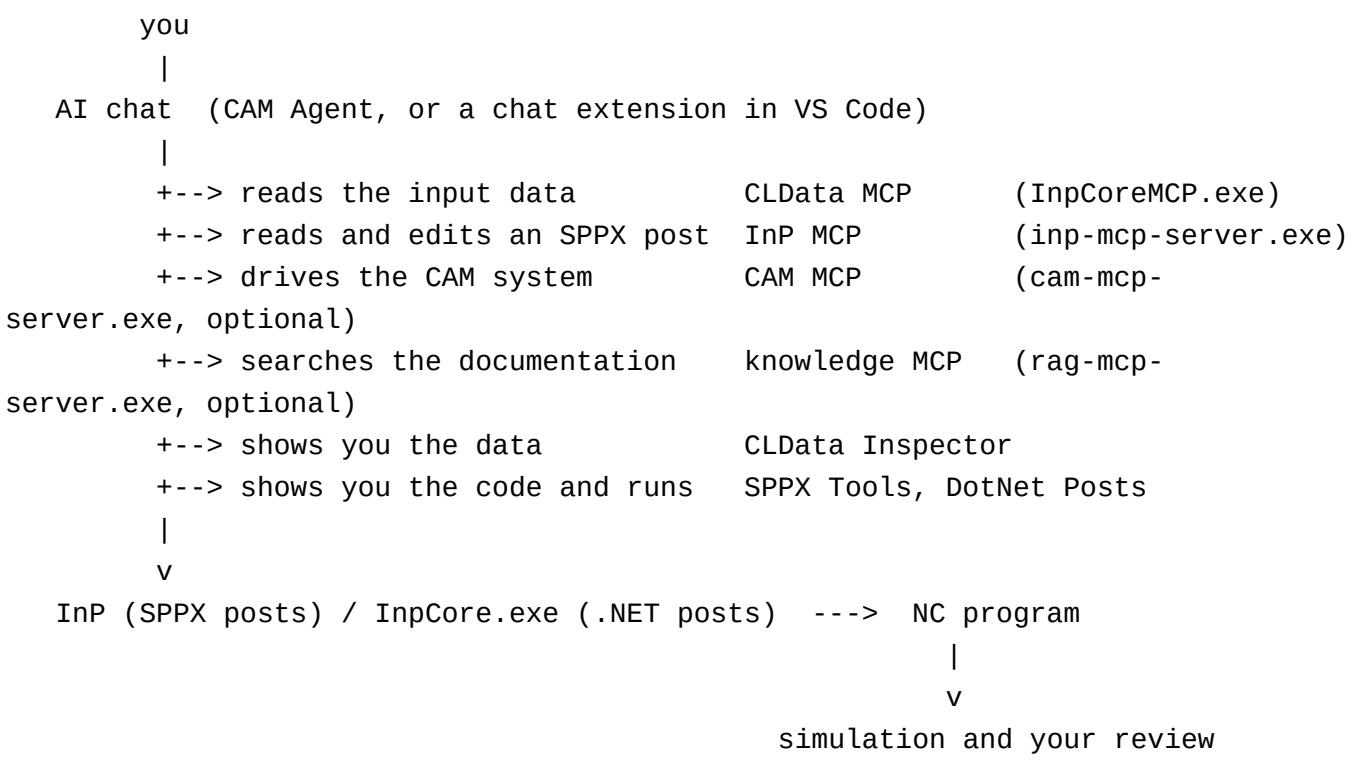
Treat every answer as a draft. The assistant sees only what you give it: it cannot know undocumented machine behaviour, and a plausible-looking NC program can still be wrong. Review the diff, compile, run the postprocessor, read the generated NC program, and verify it in simulation before it goes to a machine. Do not put credentials, customer archives or machine secrets into the chat.

How the system works

You talk to a chat. Behind it, three kinds of things do the work: servers that let the assistant read the data and drive the postprocessor, extensions that show you what it found and produced, and the postprocessor runtime that turns CLData into an NC program.

The servers are reached over MCP — the Model Context Protocol, the standard way a chat client is given access to tools. That is why the same servers work in CAM Agent and in any chat extension you connect them to.

Do not read too much into the word *server* here. These are ordinary programs on your own computer, started by the chat client when it needs them so its tools can communicate with it. They do not publish the files of your project or accept incoming network connections. The knowledge server is the exception: it queries the documentation service and keeps a local cache.



The parts

Part	What it is	What it gives you
CAM Agent	An extension of the CAM system	The chat inside the CAM system, with access to the project and the postprocessor

Part	What it is	What it gives you
Skills	Text files	Instructions in plain language that the assistant follows, so the cycle is the same every time
CLData MCP	A program on your computer, started by the chat client	Lets the assistant read the input data: files, sections, commands, parameters, machine. It never changes anything
InP MCP	The same	Lets the assistant work with an SPPX postprocessor: structure, code, registers, compile, run
CAM MCP	The same, optional	Lets the assistant look into the source project — machine, operations, tools — and prepare the input: recalculate toolpaths, export CLData, create a test project
Knowledge MCP	The same, optional; the only one that goes online	Searches a snapshot of the published documentation and of the skills — what the product does. What your team decided lives in your own notes instead
CLData Inspector	A Visual Studio Code extension	Shows you the commands and parameters the assistant is talking about
SPPX Tools	A Visual Studio Code extension	Editing support for `*.sppx` and the Generate NC panel
DotNet Posts	A Visual Studio Code extension	The same for C# postprocessors: navigation, build, run
InP	An application of the CAM system	Runs SPPX postprocessors; also the IDE you can watch

Part	What it is	What it gives you
`InpCore.exe`	A program of the CAM system, run without a window	Runs .NET postprocessors

Do not confuse these three names: `InP.exe` is the IDE and executor for SPPX postprocessors; `InpCore.exe` is the batch executor for .NET postprocessors; InP MCP is the server that lets the assistant control an InP instance. If that instance is unavailable, InP MCP reports the problem instead of silently starting a different postprocessor.

Windowed and headless

An SPPX postprocessor is executed by InP, and the assistant can start it two ways.

Windowed opens the IDE you can watch: the code it changed, the compile messages, the NC program. Use it whenever you want to follow along, and when the assistant gets stuck — you can take over in the same instance.

Headless runs without a window. It is faster and allows several parallel runs, but there is nothing on screen, so the report and the generated file are all you get.

If the assistant reports that it cannot start an instance, open InP from the CAM system yourself and let it connect.

CLData is the same for both subsystems

Both postprocessing subsystems consume the same commands and parameters; only the syntax for reading them differs. This is why the assistant should always read the actual project instead of reasoning from a command name: the documentation explains what a command means, the project says what it actually contains.

What the assistant is good at, and what it cannot do

It is good at finding the command that produced a given NC block, explaining an unfamiliar handler, drafting a small change in the style of the surrounding code, listing the cases a handler forgets, and comparing two NC programs block by block.

It cannot know machine behaviour that is absent from the data and the documentation, guarantee that generated code compiles, judge whether a motion is safe, or take responsibility for a postprocessor released to production. Compilation and a successful run prove that the software worked, not that the program is correct.

Set up the tools

The setup itself is the assistant's job. Your part is small: install the chat you are going to talk to, prepare the tools archive and a link to the setup instruction, and hand them over. Everything after that — installing the extensions, connecting the MCP servers, pointing them at the CAM installation, loading the skills — the assistant does and reports back.

1. Install the assistant

In the CAM system. CAM Agent is installed from the CAM system's own Extension Store: open the store, install CAM Agent, restart the CAM system if it asks. Nothing else is needed to start — see Develop with CAM Agent.

In Visual Studio Code. The chat is an extension like any other: open the Extensions view (`Ctrl+Shift+X`), find the one you want by name in the Marketplace, and install it. Which one to take, and what makes a client suitable for this work, is in Develop in Visual Studio Code; a client that lives in the terminal is installed its own way instead. You also need an account with an AI provider — see What it costs.

If you work only through CAM Agent, the rest of this page still applies to the knowledge and skills it uses; the tools bundle below is for the VS Code way of working.

2. Prepare the tools and instructions

The tools download address is on the External references and examples page.

- The tools bundle — an archive whose name starts with `posts-vscode-extensions-`. Download and unpack it; do not install anything yet.
- Setup instructions — open Advanced setup and give the assistant a link to that article.

If your client is already connected to a documentation service, the assistant can find the article by name through that service.

3. Hand both to the assistant

Open the chat, allow it to run commands in your workspace, and give it the task. Adjust the paths and paste:

```
Set up the postprocessor tools for me, following the instruction in
advanced-setup.md

The unpacked extensions bundle is in <bundle>
The CAM system is installed in <cam>

Install the extensions, connect the CLData and InP servers to this client,
```

```
point the extensions at the CAM installation, and load the published skills.
Then tell me what you changed and which tools you can now see.
```

Two things the assistant cannot do for you: reload the editor window, and restart the chat client after the MCP configuration changes. It will tell you when either is needed.

4. Check that it worked

Two requests in the chat are enough. Both are safe: the assistant only reads, changes nothing and runs nothing on a machine.

Reading the data. Take any CAM project — a `*.stcp` file, preferably a small test one — and write in your own words, roughly:

> Read the CLData of the project `D:\Test\Sample.stcp` and tell me the machine, the units, and how many command files there are.

A good answer names the machine, the units, and lists the files with the number of commands in each — facts from your file. Compare them with the project: if the machine is not the one you expected, you gave the wrong path.

Reading a postprocessor. Take a `*.sppx` file and write:

> Read the postprocessor `D:\Posts\Sample.sppx` and list its handlers, subroutines and registers.

A good answer is a list of names taken from that file. For SPPX the assistant needs a working InP: it will either start one itself or ask you to open the postprocessor in it.

How to spot a failure. An assistant without tools does not go quiet — it reasons instead: it lists "typical" handlers such as `OnRapid` and `OnLine`, explains how CLData is usually arranged, or asks you to paste the file contents. All of that means the tool it needed is not connected — go back to step 3, or to Advanced setup.

5. Updates

When a new version is available, the postprocessor panels show an update action — that is the only thing you need to notice. Accept it, then reload the window.

If something does not work

Troubleshooting covers the usual symptoms. Advanced setup holds the whole procedure in detail — the same page you handed to the assistant, so you can follow any step by hand and check what it did.

Organize your workspace

The assistant starts every session with no memory of the previous one. It does not remember what you agreed yesterday, which machine gave you trouble last month, or why a particular block is formatted the way it is. Whatever is written down in your working folder is what it knows; whatever stayed in the chat is gone.

So the working folder is not housekeeping — it is the assistant's memory. A few hours spent on it once saves the same explanations in every later session.

What to keep in it

One folder, opened as the workspace in VS Code:

```
work\
├── AGENTS.md      what the assistant must know every time
├── posts\         the postprocessors you work on
├── projects\      test CAM projects and reference NC programs
├── docs\          a copy of this documentation
└── notes\         your own practices and solved cases
    └── skills\    your own skills - instructions for the assistant
```

The names do not matter; the separation does. Sources apart from test data, your own knowledge apart from the official documentation.

AGENTS.md — the short standing brief

This file is read at the start of every session, so keep it short — a page at most — and put the details in `notes\`. What earns its place:

- the machines and control systems you work with, and which postprocessor serves which;
- conventions that always apply: block numbering, decimal format, comments, file naming;
- where things are: test projects, reference NC programs, your notes;
- the rules of engagement — read the actual CLData before changing code, run the postprocessor and compare with the reference before reporting anything as done, never change a production postprocessor without asking;
- constraints you have already stated once, written as you said them. An assistant reads them as instructions; otherwise it will propose the same rejected idea again next month.

Ask the assistant to draft it and then correct it — it is faster than starting from an empty file.

notes — where your experience accumulates

After every solved case, one short file: what was wrong, how it was found, what fixed it, and why that way. The "why" is what saves time later: without it the assistant cheerfully proposes the option you rejected for a good reason.

Two habits keep this useful:

- one topic per file, not one growing file — you want to find things, and a small file is cheap to read;
- describe the current state. Historical asides ("it used to be like this") rot; if you keep the folder in git, the history is already there.

When a way of working repeats — the same check, the same sequence for a family of machines — write it down as your own skill in `notes\skills\` and point the assistant at it. The published skills are ordinary Markdown files; yours look the same.

And when one of them turns out to be useful beyond your own shop, you can share it. Sharing goes through CAM Agent: open its chat and ask it to send that skill to the shared knowledge base. It goes into a review queue rather than straight into the published set, and once a reviewer accepts it, everyone using these tools gets it. Only send what you are willing to publish — a skill that names customers, part numbers or the particulars of your machines belongs in `notes\` and nowhere else.

A task is not finished when the postprocessor works. It is finished when what you learned is in `notes\`.

Documentation next to the sources

Two kinds of knowledge live side by side here, and it is worth keeping them apart.

The published documentation is a snapshot: the assistant can search it through the knowledge server, and a copy in the workspace serves the same purpose offline and at a revision you know — see Advanced setup. It answers what the product does, and it changes only when a new version is released.

Current manuals are available through the ENCY documentation center. Use the knowledge server or give the assistant a link to the relevant article. For offline work, save the required material locally and record its version and date.

Your own documentation — `notes\` above — is the living half, and the only one that knows your machines, your projects and the decisions your team has made. It grows every time you solve something. The published snapshot cannot contain it, and neither can the assistant: nobody but you writes this part.

When a documented detail contradicts the installed product, the product wins — and the note about it goes into `notes\`, which is exactly the kind of thing the living half is for.

Two ways to keep history

Way 1. Folders and copies — you keep the history yourself

You keep the files in ordinary folders and take care of their past by hand: before a substantial change you copy the postprocessor, you keep the reference NC programs next to the test projects, and dated folders serve as the history. This is a legitimate way to work, plenty of people work exactly like this, and nothing here forces you out of it.

What you pay for it is that the history depends on you remembering. Forget one copy and the previous state is gone, and questions like "what exactly changed since Friday" or "when did this break" have no precise answer any more.

Way 2. Git — a tool keeps the history for you

The same folder, but the past is looked after by a tool made for exactly that, and used by software developers everywhere.

It also settles the worry that comes with letting an assistant edit your files: nothing it does is irreversible. Every state is kept, so a change that turns out to be wrong — or a whole session that went the wrong way — costs you a minute to undo, not a working postprocessor. You can let it try things it would be a pity to try otherwise.

What it gives you in this work:

- an exact answer to *what changed* between two states, down to the block;
- an exact answer to *when* a behaviour broke, and *what else* changed in that step;
- the *reason* a line exists — the message of the change that introduced it;
- a safe way to try something risky and drop it without traces;
- a way to hand a colleague the postprocessor together with its history.

You do not have to learn git for this. Ask in plain words and let the assistant do it:

- "commit this, the offset fix works" — it commits, with a message that says why;
- "compare with what we had on Friday" — it shows what changed;
- "find when the coolant block disappeared" — it searches the history;
- "why is this line here?" — it finds the change that added it and reads its message;
- "put it back the way it was before the last change" — it restores from history instead of retyping.

If git is not installed, ask the assistant to install it — the official download is at https://git-scm.com/downloads. Ask it to set the workspace up as well: one folder under version control, with the generated NC output and temporary files excluded.

When the open folder has to be the postprocessor

For a .NET postprocessor, VS Code has to be opened on the postprocessor's own folder — that is what makes the C# language service, the build and the debugger work. The standing brief and your notes then lie outside the folder that is open. Two arrangements are enough:

- Keep everything in one repository. Let `work\` as a whole be the repository and keep the postprocessor folders inside it. Git then works from any folder below it — it finds the repository that contains the folder you are in — so the history is one, whichever folder is open.
- Name the brief once in the client's settings. Chat extensions read the standing brief in the folder they were opened on and do not look above it — but it can be pointed at explicitly. Kilo Code has a global `instructions` setting in its `kilo.jsonc` file (usually `~/.config/kilo/kilo.jsonc`): it lists paths to additional instruction files, and they apply across all projects. Put the full path of your `work\AGENTS.md` there, and it will be read whatever folder you open.

  If your client cannot do that, leave a short `AGENTS.md` in the postprocessor folder instead — two lines: the rules are in `..\..\AGENTS.md`, the notes in `..\..\notes\`.

Skills are easier: installed for the user rather than for one project, they are available whatever folder is open, so they only have to be put in place once.

For SPPX none of this is needed: `work\` can be opened as it is.

What must not go in

Passwords, tokens, licence files, customer archives that have nothing to do with the postprocessor, and unrelated repositories. Everything in the workspace is material the assistant may read and, if you keep history, material that stays in it.

Develop with CAM Agent

CAM Agent is the chat inside the CAM system. It is the shortest path from a task to a checked result: you describe what the NC program should look like, and the agent investigates the project, changes the postprocessor, compiles it, runs it and reports back. You do not have to open the code — although for SPPX you can watch every step in the postprocessor IDE if you want to.

Install it

CAM Agent is a regular CAM system extension: open the Extension Store in the CAM system, install CAM Agent, and restart the CAM system if the store asks for it. Its own documentation, available from the store entry, covers the chat itself; this page covers only postprocessor work.

An introductory allowance comes with the agent, so the first sessions need nothing else from you; its size and conditions may change over time. After that, connect your own account with an AI provider in the agent's settings and the work continues on that provider's terms — see What it costs.

The working cycle

1. Open a project that contains a short, representative piece of the toolpath. A small test project makes every following step faster and the comparison readable.
2. Open the postprocessor you want to change, or tell the agent which file to use.
3. State the task in terms of the NC program: which block is wrong, what it should look like, on which machine. "Add M08 after the tool change" is a workable task; "make the coolant right" is not.
4. Ask for a plan first, and for the data it is based on. The agent should name the CLData command it found and the handler that processes it before it proposes an edit.
5. Approve the change. The agent edits the handler, compiles the postprocessor, runs it on the same project and returns the generated NC program.
6. Read the report: the changed code, the compile messages, and the NC blocks that differ from the previous run.
7. Verify the result as described in Review and verify. Then either accept it or refine the task.

Steps 4 and 5 are where the time is saved or lost. An agent that has read the actual data produces a small, explainable edit; an agent that was not given a project guesses from the command name.

What you see while it works

For SPPX the agent drives the postprocessor IDE. Started windowed, the IDE is on screen: the handler it edited, the compile messages, the generated NC program. This is the mode to use when you want to follow along, and it is also the mode to use when the agent gets stuck — you can take over in the same instance.

Started headless, InP runs without a window. It is faster and it can process several runs in parallel, but there is nothing on screen, so the report and the generated file are all you get.

For .NET postprocessors the agent works with the project and the batch runner, so there is no IDE to watch; you see the build result, the run report and the NC file.

Knowledge

CAM Agent answers from an indexed copy of the product documentation and of the official skills, with an offline cache, and it keeps your own skills alongside them. It does not clone the documentation repository into your project; if you want the agent to read a specific local copy of the documentation, say so explicitly and give the path.

This is also where a skill of your own is shared: ask CAM Agent to submit it, and it goes into the review queue for the published set — see Organize your workspace.

Limits

The agent does not replace a postprocessing engineer. Two rules are worth stating plainly: do not let it change a production postprocessor without your review, and do not let it send an NC program to a machine. A run that ended with code 0 means the tools worked, not that the program is safe.

Develop in Visual Studio Code

In VS Code you stay in control of the code: you see every diff, run the generator yourself when you want to, and keep the postprocessor in version control as usual. The assistant is a chat extension of your choice; the results are shown by the three extensions from the bundle.

Install everything first — see Set up the tools.

Which chat extension to use

Whichever you pick, it runs on your own account with an AI provider and is billed by that provider from the first request — see What it costs.

Any assistant that meets three conditions will do, and the choice is yours:

- it connects to MCP servers, which is how it reads CLData and drives the postprocessor;
- it may run commands in your workspace, which is how it builds, installs and opens things for you;
- it has a place for standing instructions, so the rules of your project apply to every request.

The recommended starting point is Kilo Code. Three of its properties matter for this work:

- Skills are a first-class mechanism. A skill is a folder with a `SKILL.md` file, and Kilo Code picks it up from `.kilo/skills/` in the project or `~/.kilo/skills/` for all projects — which is exactly the shape of the skills published with this guide, so they drop in unchanged. New skills are discovered when a session starts, or on `/reload` in an open session.
- It is not tied to one model provider. Requests go through a gateway that reaches models from every major provider through one endpoint, and you can bring your own API keys, so the choice of provider stays yours.
- It is open source (MIT-licensed), and its sessions can be monitored and steered remotely, including from a phone.

Others that work with this toolset:

Chat extension	Standing instructions	Notes
Kilo Code	`AGENTS.md` in the project root; project instructions in `kilo.jsonc`	Loads the published skills as they are
Cline, Roo Code	The client's rules file	Same origin, so the panels look familiar

Chat extension	Standing instructions	Notes
Claude Code	`CLAUDE.md`; loads the published skills as they are	Runs commands after you approve them
GitHub Copilot Chat	`.github/copilot-instructions.md`	MCP and command execution require its agent mode
Codex CLI	`AGENTS.md`	Works from the terminal, alongside VS Code

Whichever you choose, prefer a client that loads skills directly — you then get the way of working as well, not just the tools. With a client that has no skill mechanism, name the relevant `SKILL.md` in the standing instructions or paste it into the chat; the content is plain Markdown and works either way.

Where each client keeps its MCP configuration is listed in Advanced setup.

The working cycle

1. Work on a copy or a branch of the postprocessor, never on the production file.
2. Open only the folders that matter: the postprocessor, and the documentation checkout if you use one.
3. Open a small test project. Ask the assistant to inspect it — the CLData Inspector then shows you the same commands it read.
4. State one bounded task, with the expected NC output. Ask for a plan and the evidence behind it before any file is written.
5. Review the diff as soon as it appears. Reject reformatting and rewrites you did not ask for.
6. Let it compile and run, or run Generate NC yourself. Compare the generated program with the previous run and with the reference program.
7. Verify the result — see Review and verify.

For the details of each postprocessor type, continue with the SPPX workflow or the .NET workflow.

Ask it to show you things

The assistant can open the exact item it is referring to, so you see the current source or input data rather than a fragment quoted in chat:

- a command in the CLData Inspector — the exact file, command and parameter;
- a handler at the relevant line, with a range highlighted and a short note attached;
- the Generate NC panel with a configuration prepared, and the run started if you asked for it.

Just ask: "show me that command in the inspector", "open the handler where you changed the offset".

Note that the assistant's own run is silent — it runs the postprocessor through the server and reads the output itself, which is how it checks its own work. The Generate NC panel is for you: the configuration, the post input parameters, the progress, the errors in Problems with links into the handler code, and the diffs against the previous run and the reference program.

Let the assistant keep its own run configuration

The Generate NC panel stores named configurations: which project to process, where to write the NC file, which post input parameters to use. Ask the assistant to create its own — for example `Agent check` — instead of reusing yours. Your settings then stay untouched, and the two runs can be compared.

Context and safety

Give the assistant a small set of authoritative sources: the handler in question, the relevant page of the CLData reference, a reduced test project, and a known-good NC program. A whole customer project rarely improves the answer.

Use the workspace instruction file (`CLAUDE.md`, `AGENTS.md`, `.clinerules`, depending on the client) to state the conventions that always apply: naming, output format, the machines involved, the requirement to run the postprocessor before reporting success. Do not put credentials, tokens or customer archives there, and keep generated NC output out of the source folders.

Ask for a plan before a multi-file change, keep every change in version control, and never let an assistant transmit a program to a machine.

What the extensions give you

Three extensions do the work you see: one for editing and running SPPX postprocessors, one for looking at the input data, one for C# postprocessors. The assistant drives the same commands, but they are ordinary VS Code features — this page is what to expect from them and where to find it.

Commands are run from the command palette: `F1` (or `Ctrl+Shift+P`), then start typing the name. Commands whose title ends with (agent API) exist for the assistant; you never need them by hand.

SPPX Postprocessor Tools

Editing

Open a `*.sppx` file and you get the XML structure and the handler code highlighted as two different languages. The XML wrapper is folded away, so the file reads as a list of handlers with their code. The Outline view lists handlers, subroutines and registers and navigates to them, `F12` jumps to a definition, completion and signature help cover the built-in language, and syntax errors are underlined as you type.

If VS Code restores an old folding layout and the file looks wrong, run SPPX: Unfold Code (reset folding to default layout).

Generating an NC program

The Generate NC panel is the main working surface. With a `*.sppx` file open, it opens from the triangular play button in the editor title bar — hover it and the tooltip reads SPPX: Generate NC Program — or from the command of the same name in the palette.

In the panel:

- named run configurations — the CAM project to process, the output file, and the postprocessor's own input parameters, shown with their captions rather than internal names;
- Run, with progress while the generation goes on;
- errors and warnings in the Problems view, each linking to the line in the handler that produced it;
- the generated program, with diffs against the previous run and against a reference program you point at.

Buttons next to Run open the CLData of the current configuration in the Inspector — the project name opens it in the sidebar, the arrow in a panel below the editor. The same is available as SPPX: Inspect CLData of the Current Run Configuration.

Two more commands are worth knowing: SPPX: Open in Postprocessors Generator IDE opens the same postprocessor in InP, and SPPX: Clean Stored Run Data discards the stored run state if a configuration gets into a bad shape.

CLData Inspector

This is where you see what actually arrives at the postprocessor. Its icon in the activity bar opens three views:

- Files — the files of the opened project with their command counts. The view's buttons open a CAM project (`*.stcp`, `*.stc`) or a single `*.inpcld` file, reopen a recent one, reload the project after it was regenerated, and open the wide inspector tab in the editor area.
- Commands — the commands of the selected file, grouped by structure sections; in NC-subroutine files, by subroutine boundaries. `Ctrl+F` in the view, or CLData: Find Command⋯, searches on the server side, so it stays usable on files with hundreds of thousands of commands.
- Parameters — `Key = Value` for the selected command: the raw array, the named parameters, and the typed .NET view. CLData: Toggle .NET Parameter View switches the last one on and off.

Right-click a parameter to copy what you need for code: Copy Name (sppx) and Copy Name (.NET) give the exact form to paste into a handler, and Copy Value the value itself. On a command, Copy Caption copies its human-readable name.

The assistant can open a specific file, command or parameter here when it wants to show you something — see Develop in Visual Studio Code.

DotNet Posts

For C# postprocessors, with the same Generate NC panel as SPPX Tools — configurations, input parameters, progress, results and diffs.

Two ways to start a generation:

- DotNet Posts: Generate NC Program, or the triangular play button in the editor title bar of a C# file (its tooltip carries the same name) — works with the postprocessor project open in the workspace and finds the built assembly itself;
- DotNet Posts: Generate NC Program from DLL⋯, or a right-click on a `.dll` in the Explorer — runs a compiled postprocessor without any sources. Several postprocessors can be kept in separate panel tabs and compared on the same project.

DotNet Posts: Reveal Postprocessor Handler opens the handler that processes a given command. The `dotnetPosts.autoBuild` setting builds the project before each run, so a change is never tested against a stale assembly.

Postprocessor Tools updater

The updater has no interface of its own. When a new version is available, the postprocessor panels show an update action; accept it and reload the window. The same check is available from the palette — see Advanced setup.

Settings

Each extension's settings page starts with the folder where the CAM system is installed — that is the one setting that has to be right for anything to work. Advanced setup lists all three and the picker that fills them in.

SPPX workflow

An SPPX postprocessor is a set of handlers — one per CLData command — plus subroutines, objects and registers, written in the built-in Pascal-like language, with output formatting described by masks. The assistant works on exactly these items through the InP MCP server, and the cycle is always the same: read, change, compile, run, compare.

For the language and the mask reference, see the Postprocessors generator documentation; for the meaning of the input commands, see the CLData reference.

The cycle

1. Prepare. Work from the closest postprocessor that already exists — the one in use for this machine, or a distributed one for a similar control system — rather than from an empty file. Copy it, or commit the current state, so the comparison has a baseline. Generate the NC program once before any change and keep it.
2. Find the data. Identify the CLData command behind the NC blocks you want to change, and read its actual parameters from the test project. The command name alone is not enough: the same command carries different values in different projects.
3. Find the handler. The postprocessor structure lists handlers, subroutines and objects. The handler named after the command is where processing starts, but the output may be produced further down — in a subroutine, a mask, or on the next modal state change.
4. Read before writing. The assistant must read the current source of the item it is about to change, and the register definitions if registers are involved. Editing what it has not read is how unrelated code disappears.
5. Change the minimum. Preserve register order and formatting, modal behaviour, separators, the state kept in `Common` and local variables, and the existing subprogram conventions. Adapted donor code from another postprocessor has to be checked against this project's data, not pasted.
6. Compile. Every reported error must be fixed before going further; there is no point in running a postprocessor that did not compile.
7. Run. Interpret the same test project and get the NC program. A change that was never run is not finished, whatever the code looks like.
8. Compare. Diff the new NC program against the baseline, then against the reference program from the machine. Trace every difference to a command and a handler — including differences you did not expect.
9. Verify. Continue with Review and verify.

What the assistant can and cannot change

There is one handler per CLData command: handlers can be edited but not created or deleted. Subroutines and objects can be created. Deleting an item is irreversible, so the assistant asks first — read the request before you confirm.

Watching it work

Ask for a windowed InP instance when you want to see the IDE: the handler it edited, the compile messages and the generated program are in front of you. A headless instance is for batch checks and parallel runs — nothing appears on screen. With several InP windows open, tell it which one to work in.

In VS Code you can also ask it to open the changed handler in the editor, or to prepare a Generate NC run so you see the panel, the parameters and the diffs yourself.

What to check in the code

Register formatting and modal output are where AI-drafted handlers usually go wrong: a value written unconditionally instead of on change, a modal register left set, an axis word emitted with the wrong sign or in the wrong order. Review every conditional branch and every direct output statement, and test the values the fixture does not contain — omitted parameters, repeated commands, boundary and maximum values, and the transitions at program start and end.

Do not let .NET property names appear in SPPX code. The two subsystems read the same data with different syntax: the CLData pages describe the meaning, the SPPX pages describe the access form, and index bases differ between them.

.NET workflow

A .NET postprocessor is a C# project built against the postprocessing SDK: typed handlers, a `.csproj`, and a compiled assembly that the batch runner executes. The assistant works here as it does in any C# project — with the C# extension providing the language service and `dotnet` doing the build — and the postprocessor-specific parts are the template, the settings file and the runner.

For the API, see the .NET postprocessors documentation; for the input data, see the CLData reference.

Where to start

Most of the time, not from scratch. Take the closest postprocessor that already exists — one supplied with the CAM system, or one written earlier for a similar machine or control system — copy it, and adapt it. You start from code that already works, and the differences you have to reason about are the ones that actually matter.

If a new postprocessor really is needed, create it in the CLData Viewer: CLData viewer → Create new postprocessor → Simple postprocessor⋯, then pick an empty folder — its name becomes the name of the postprocessor. The project is created for you, and its context menu opens it in VS Code. The .NET postprocessors documentation describes this in "How to prepare the computer to start writing the postprocessors". Ask the assistant only if you want it done from the templates instead — that is its job, not yours.

When you copy a .NET postprocessor into a folder with a new name, rename its `.csproj` file to the same name. The folder name and the `.csproj` name must match: the template build task takes the project path from the folder name, and the debug configuration takes the assembly name from it. If they differ, both configurations look for files that do not exist.

Work on it with VS Code opened on the postprocessor's own folder — the C# language service, the build and the debugger expect that. Keeping your standing brief and your notes reachable from there takes one arrangement, described in Organize your workspace.

Either way, start from a known state:

1. Put the postprocessor under version control before changing anything.
2. Build it unchanged. If it does not build as it is, no later diagnosis is worth anything.
3. Generate an NC program from it and keep that as the baseline for comparison.

The cycle

1. Inspect the test project's CLData and find the command you need. The Inspector shows every command in the typed .NET projection as well — the interface, the handler name and the property values — and can copy a parameter name in the form you paste into C# code.

2. Locate the typed handler for that command. Ask the assistant to list the handlers of the project and open the relevant one.
3. Make one small change, with the expected NC output stated in advance. Use the named typed properties from the SDK rather than raw indexes when the SDK provides them.
4. Build. Treat warnings as information about your change, not as noise to be silenced.
5. Run on the test project and read the run report: the NC file it produced, the messages, the result code.
6. Compare the generated program with the baseline and continue with Review and verify.

Running from VS Code

DotNet Posts drives the run and reports it back in the Generate NC panel, the same way SPPX Tools does. Two entry points cover both situations: Generate NC Program works with the postprocessor project open in the workspace and finds the built assembly itself, and Generate NC Program from DLL… takes a compiled postprocessor you point at, without any sources — which also means you can keep several postprocessors in separate Generate NC tabs and compare their output on the same project.

`dotnetPosts.autoBuild` makes the extension build the project before a run, so an edit is never tested against a stale assembly. The panel also has the commands the assistant uses to work with you: listing and opening handlers, highlighting a range of code, reporting where your cursor is, and opening the current CLData in the Inspector.

Running a postprocessor you have no sources for

A postprocessor carries its default settings inside the assembly, and the runner can write them out as a `Settings.xml` next to the input — which is what makes Generate NC Program from DLL… possible. Use the file the runner produced; the format is not the same as the SPPX one, so a hand-written file will not do. If settings cannot be extracted, the installed CAM system is older than this workflow needs — check its version before suspecting the postprocessor.

A run counts as finished only when the report and the process result agree. A run that ended silently produced nothing worth comparing.

What to check in the code

Review null handling and boundary values, units, culture-sensitive number formatting, the encoding and line endings of the output file, modal state kept between commands, tool compensation, coordinate transformations, and the exception paths. Generated C# has to be built and exercised on representative data before it is trusted — for a typed API the compiler catches a smaller share of the real mistakes than it appears to.

Review and verify

AI assistance changes how a draft is produced. It does not change how a postprocessor is accepted. A successful build and a clean run prove that the tools worked — nothing more.

Say how far the check went

Use one of four states, and keep the evidence for it. The point is to make an unfinished check visible instead of calling everything "done".

State	What it means
Not verified	There is a plan, a diff or a build, but the postprocessor was not run on a recorded project
Technically verified	The run completed as expected and the log, result code and generated file are kept; the content of the NC program has not been reviewed
Verified on a test project	The baseline and the new NC program were compared on the normal test operations and applicable boundary cases — such as program start and end, operation or tool transitions, repeated modal commands, and minimum or maximum parameter values — and the machine parameters were checked
Approved for production	A qualified engineer confirmed the program for a specific machine, control system and application

Only the last state allows production use, and only a person can grant it. Never describe a result as ready or safe on the strength of a process exit code.

Review the source change

- The diff contains the requested change and nothing else — no reformatting, no unrelated rewrites.
- Command names, parameters, units, coordinate systems and modal assumptions match the CLData reference and the actual project data.
- The postprocessor compiles, and the compile messages were read rather than skipped.
- Normal, omitted, repeated, maximum and unsupported inputs were exercised, not only the one case in the fixture.
- Output file name, encoding, line endings and error handling are as the machine expects.

Review the NC program

Compare against a known-good program and start at the first differing block; a difference early in the program often explains everything after it. Trace each difference back to the CLData command and the handler that produced it.

Check the program as a whole:

- the selected machine, control system, units and work coordinate system;
- program start and end, including the final positioning and program stop;
- tool changes, spindle, feeds, coolant and safe positions;
- signs, axis order, compensation, work offsets and retracts;
- cycles, subprograms, rotary motion, limits and synchronization;
- no unexpected motion, no missing commands, no leftover blocks;
- the program still corresponds to the original toolpath, on the test section and on the boundary cases.

Then run it through the simulator and through the machine-specific verification you normally use. Investigate warnings instead of suppressing them.

Keep the evidence

Store the source diff, the input project, the postprocessor version, the run report and the generated program together, and record who reviewed it and when. When a problem surfaces later, this is what makes it possible to tell a postprocessor change from a data change.

What the assistant must not do

It must not approve code, change machine configuration to make output pass, bypass a check, or transmit a program to a control system. Human approval stays between the generated output and production use.

Troubleshooting

The assistant does not see the postprocessor tools

Ask it which tools it currently has. If the CLData or InP tools are missing, the MCP configuration did not load: check that you edited the file your client actually reads, that the paths are absolute and exist, that backslashes are doubled in JSON, and restart the client. A client started before the configuration was saved keeps the old set.

The assistant cannot reach InP

InP MCP drives a running instance of the postprocessor IDE. Let the assistant start one itself, or open InP from the CAM system with the postprocessor loaded and ask it to connect. If it reports that the instance manager is unavailable, the installed CAM system does not support starting instances — open InP by hand.

A headless instance that fails to compile or run with an error about focusing an invisible window is an older InP build. Use a windowed instance instead.

Several InP instances are running

The assistant has to address one of them explicitly, and a request that does not is rejected rather than sent to an arbitrary instance. Close the instances you do not need, or tell it which one to work in — this is normal when you deliberately run several postprocessors in parallel.

Command indexes no longer match the data

CLData indexes are valid for one revision of the data. When the project is regenerated, the server rereads it and asks for the call to be repeated; results captured earlier point at different commands. Always keep the project together with the indexes, and repeat the inspection after regenerating CLData.

An extension says the installation folder is not set

Set it in the extension's settings, or run its Select Installation Folder··· command from the command palette (`F1`) and pick the folder. Both the product folder and its `Bin64` subfolder are accepted, and quotation marks around a pasted path are tolerated. With several CAM versions installed, make sure it points at the one you actually want to test against — this is the most common cause of "it works differently than in the CAM system".

The result does not match what the CAM system produces

With more than one CAM version installed, the run may have gone through a different one — nothing warns you, because it succeeds and simply produces different output. Check that all three extension settings point at the version you are developing against.

For a .NET postprocessor, the extension keeps debugging on the same installation as the panels, and says so when it has to override the `InpCoreDir` environment variable. If you run the postprocessor outside the extension — from a terminal, or from a configuration of your own — that variable decides instead, and it holds whichever installation's CLData Viewer started last. Changing it does not affect programs that are already running; signing out of Windows and back in is what makes a new value take effect everywhere.

Generate NC produces no file

Read the errors in Problems first: a postprocessor that failed to compile or stopped on a run-time error produces nothing. Then check that the output path is writable and that the configuration names the project you think it does. For .NET, check that the assembly was rebuilt after the last edit — `dotnetPosts.autoBuild` does that for you.

Settings cannot be extracted from a .NET postprocessor

Extracting the default settings from the assembly requires a recent batch runner. If the installed one does not support it, use a project that has its settings file, or update the CAM system; do not hand-write a settings file to work around it.

install.cmd closes before I can read it

It waits for a keypress, so a window that closed immediately means it was not the script that ran. Run it from a command prompt in the unpacked folder to keep the output visible.

If it reports that the `code` command was not found, the VS Code command-line launcher is not on `PATH`. The VS Code installer normally adds it; otherwise add the folder containing it to `PATH`, or install the `.vsix` files through Extensions → Install from VSIX⋯.

The updater does not offer anything

Updates are configured by the bundle, so extensions installed by hand from separate `.vsix` files are never offered any. Reinstall with `install.cmd` from the unpacked archive, then run Postprocessor Tools: Check for Updates.

The assistant answers from documentation instead of the project

This is what happens when the tools are unavailable, when the request named no project, or when the answer is not required to cite the data. Give it the project, ask which command and which parameter values it actually read, and let the skills do their job — they require inspection before editing.

The assistant reports success without running anything

Compilation is not a result. Ask for the generated NC program and the comparison with the baseline; if there is none, the state is Not verified — see Review and verify.

The documentation the assistant reads is outdated

A local checkout is a snapshot: update it with `git pull` and note the revision when it informs a release decision. Version-sensitive details — SDK versions, command names, panels — should be confirmed against the installed product rather than the checkout.

Advanced setup

This page is the full setup procedure and the reference for checking what was done. Hand it to the assistant as described in Set up the tools, follow it by hand when automatic setup is unavailable, or read it when a team needs repeatable results.

What this is about

If you are an assistant reading this page to set the tools up, here is the ground you are standing on.

A CAM system calculates the toolpath and writes it out as CLData — a stream of commands and parameters that describes the machining independently of any particular machine. A postprocessor turns that stream into the NC program for one specific machine and control system: its codes, its formats, its conventions. Every machine on the shop floor needs its own, and adapting one is the daily work of the person you are helping.

There are two kinds of postprocessor, and they never mix:

- SPPX — a set of handlers, one per CLData command, written in a built-in Pascal-like language, executed by InP;
- .NET — a C# project built against the postprocessing SDK, executed by `InpCore.exe`.

Both read the same CLData. That is why reading the real project data always comes before changing code.

What you are being set up to do: read the input data through the CLData server, read and change a postprocessor through the InP server or the C# project, compile it, run it on a real test project, and compare the generated NC program with a known-good one. The VS Code extensions are the human's window into all of that — they show the data you found, the code you changed and the program you produced. The person you work with decides whether the result is correct; you produce the evidence they judge it by.

So the setup has four parts, in this order: the extensions, the folder each of them needs, the MCP servers, and the knowledge — this documentation plus the skills. After that, organise the workspace itself: see Organize your workspace, and follow the `prepare-workspace` skill.

Record the versions when you report a problem: the CAM system, the extensions bundle, CAM Agent, the MCP servers and the AI client. Most "it worked yesterday" reports are version differences.

Install the extensions

Visual Studio Code 1.85 or newer is required. In the unpacked bundle, `install.cmd` installs every `.vsix` next to it and replaces older versions; it needs the `code` command on `PATH`, which the VS Code installer adds by default. Without the script, install each file with `code --install-extension <file>.vsix`, or

through Extensions → Install from VSIX⋯ in VS Code. Then reload the window: `F1` (or `Ctrl+Shift+P`) opens the command palette, where Developer: Reload Window is run. Every command written in bold with a colon on this page is a VS Code command, run the same way.

The bundle holds four extensions: SPPX Postprocessor Tools, CLData Inspector, DotNet Posts and the updater.

Point the extensions at the CAM installation

Each extension needs the folder where the CAM system is installed, because that is where the executables it drives live. This is the first setting on each extension's settings page:

Setting	Needed by	Executable
`sppx.installationFolder`	SPPX Tools	`InP.exe`
`cldata.installationFolder`	CLData Inspector	`InpCoreMCP.exe`
`dotnetPosts.installationFolder`	DotNet Posts	`InpCore.exe`

Both the product folder and its `Bin64` subfolder are accepted, and a path pasted with quotation marks is accepted as well. Instead of typing it, run the extension's command — for example DotNet Posts: Select Installation Folder⋯ — and pick the folder in the dialog; the settings page offers the same picker as a link. SPPX Tools and CLData Inspector borrow each other's value while their own setting is empty, so in a normal installation the path is set once.

Connect the MCP servers

MCP is what gives the assistant the ability to act. Any MCP-capable client works, and the configuration is the same everywhere except for the file it goes into.

CLData MCP — `InpCoreMCP.exe`, in the `Bin64` folder of the CAM installation. It works on its own: it opens a project (`*.stcp`, `*.stc`) or a single `*.inpcld` file and needs no running CAM system, and it never modifies the data.

InP MCP — `inp-mcp-server.exe`, shipped with CAM Agent; in a default installation `%LOCALAPPDATA%\CamAgent3\bin\inp-mcp-server.exe`. It drives the postprocessor IDE, and can either start an instance itself, windowed or headless, or attach to one you opened from the CAM system.

CAM MCP — `cam-mcp-server.exe`, next to the previous one. Optional, and not a postprocessor tool: it drives the running CAM system. Connect it when the answers you need are in the source project rather than in the CLData — the machine and its schema, the parameters of an operation, the tools and their properties, the coordinate systems and the workpiece setup. It can also produce the input you

postprocess: recalculate the toolpaths, export the CLData of the active project, or create and open a test project. Useful when a test case has to be built or adjusted rather than merely read.

```
{
  "mcpServers": {
    "inpcld": {
      "command": "C:\\Program Files\\<CAM installation>\\Bin64\\InpCoreMCP.exe",
      "args": []
    },
    "inp": {
      "command": "C:\\Users\\<user>\\AppData\\Local\\CamAgent3\\bin\\inp-
mcp-server.exe",
      "args": []
    },
    "cam": {
      "command": "C:\\Users\\<user>\\AppData\\Local\\CamAgent3\\bin\\cam-
mcp-server.exe",
      "args": []
    },
    "knowledge": {
      "command": "C:\\Users\\<user>\\AppData\\Local\\CamAgent3\\bin\\rag-
mcp-server.exe",
      "args": []
    }
  }
}
```

The first two are what postprocessor work runs on; `cam` and `knowledge` are useful additions — leave out what you do not need.

Use absolute paths, and mind that JSON needs backslashes doubled. Where the configuration goes depends on the client:

Client	Where the configuration goes
Kilo Code	`kilo.jsonc` in the project root or `.kilo/kilo.jsonc`; globally `~/.config/kilo/kilo.jsonc`. The UI writes it for you: settings icon → Agent Behaviour → MCP Servers

Client	Where the configuration goes
Roo Code	`.roo/mcp.json` in the project, or the global `mcp_settings.json` — MCP Servers → Edit Project MCP / Edit Global MCP
Cline	`cline_mcp_settings.json` — MCP Servers → Configure
Claude Code	`.mcp.json` in the project root, or `claude mcp add`
Copilot Chat in VS Code	`.vscode/mcp.json`, `servers` section
Codex CLI	`~/.codex/config.toml`, section `[mcp_servers.inpcld]` with `command = "…"`

Where a client supports both levels, the project file takes precedence over the global one. The client has to be restarted after the file changes. Such a file contains local absolute paths, so keep it out of a shared repository.

These are plain executables on the same computer: the client starts one and talks to it directly over its standard input and output, so none of them needs a port, a service or an account of its own — the word "server" says nothing about the network here. That is why one configuration fits any client, and why a server that does not answer is a path or a permission problem. The knowledge server is the exception that proves it: documentation search goes out to an online service, while everything it has cached keeps working without one.

Check the connection

Use a non-production project and a small read-only request for each server: the files of a CLData project, and a ping of InP with the structure of the open postprocessor. Compare the answers with the project you named. Never include a write, a source edit, an NC transmission or a machine connection in a connectivity test.

Documentation and skills for the assistant

Besides the tools, the assistant needs the reference material.

Two sources, and they answer different questions.

- The knowledge server — `rag-mcp-server.exe`, in the CAM Agent folder next to the other servers. Add it to the MCP configuration the same way; it needs nothing else. It holds a snapshot of the published documentation — the product as a whole and postprocessing in particular — and searches it by keyword and by meaning, reads a page in full, and lists and searches the published

skills. It can also index a CAM project so that its operations, tools and machine become searchable. This is what CAM Agent answers from, and the same server works for any MCP client. Being a snapshot, it is static by nature: it tells you what the product does, not what your team decided. It keeps a local cache, so ask the assistant to refresh it once — after that the skills are searchable offline.

The published set can also grow from your side: a skill you wrote yourself can be submitted for review and, once accepted, becomes part of it for everyone. Submitting is done from the CAM Agent chat, and only for material you are willing to publish.

- The workspace documentation — your own, and it grows. Your projects, the techniques that work on your machines, the conventions your team has settled on, the cases you have already solved. This is the part the assistant cannot get anywhere else, and the part that makes its answers yours rather than generic. How to organise it is a topic of its own: Organize your workspace.

  Open the current manual through the ENCY documentation center. Give the assistant the relevant article link or use the knowledge server. If you need an offline copy, save the material with its version and date.

The skills published with this guide live in the `skills` folder of this documentation module — each one a folder with a `SKILL.md` file inside it. Take the whole set; it grows with the toolset, and an assistant only uses the skill that fits the task in front of it. In Kilo Code, copy the folders into `.kilo/skills/` in the project, or into `~/.kilo/skills/` (`%USERPROFILE%\.kilo\skills\` on Windows) to have them in every project; a new session picks them up, and `/reload` does it without restarting. In other clients it depends on the client — a workspace or user skills folder, an agent configuration entry, or manual inclusion. If your client has no skill mechanism, paste the relevant `SKILL.md` into the chat or reference it from the client's instruction file (`CLAUDE.md`, `AGENTS.md`, `.clinerules` and similar). CAM Agent manages its own set and keeps yours alongside it.

Whichever way they are provided, check that they are loaded: ask the assistant which skills and tools it currently sees.

Updates on demand

The panels offer an update action when a new version is available. The same check runs from the palette: Postprocessor Tools: Check for Updates, then Postprocessor Tools: Install Available Updates, then Developer: Reload Window. Updates are configured by the bundle, so extensions installed by hand from separate `.vsix` files are never offered any.

Several CAM installations on one computer

Each extension points at one installation folder, and the assistant runs whatever is in it. With a released version and a beta version side by side, decide which one the postprocessor is being developed against and set all three settings to it. When you switch, switch all of them — a mixed configuration where

CLData is read by one version and the postprocessor is run by another produces results that cannot be reproduced.

There is one path the settings do not own: the `InpCoreDir` environment variable, written by the CLData Viewer when it starts, so it holds whichever installation's viewer ran last. The debug configuration that comes with the .NET template is written in terms of that variable, but DotNet Posts substitutes the folder from its own setting when a debug session starts and warns once when the two disagree — so debugging follows the same installation as the panels. The variable is used only when the setting is empty.

Changing the variable is not instant, which is worth knowing when you do rely on it: programs already running keep the old value, and in practice it takes signing out of Windows and back in for a new one to reach everything.

Pin the documentation

For work that has to be reproducible, keep the documentation checkout at a known revision instead of pulling it constantly, and record that revision together with the postprocessor version. A retrieval service always answers from the current index, which is convenient for exploration and unhelpful when you need to explain why an answer changed.

Give the assistant its own workspace

- Its own run configuration in the Generate NC panel, so your settings stay as you left them.
- Its own output folder for generated NC programs, outside the source tree.
- A branch or a copy of the postprocessor, never the production file.
- A reduced test project instead of the full customer project. Smaller data means a readable diff and a faster cycle.
- No credentials, tokens, customer archives or unrelated repositories in the workspace or in the instruction files.

Automated checks

A headless run is suitable for a repeatable check: a fixed input project, a fixed postprocessor revision, captured logs and a stored NC program. Two rules keep it useful. Run it in a disposable working folder, so a failed run cannot leave a half-modified postprocessor behind. And review the diffs and the NC output outside the automation — an automated pipeline can tell you that output changed, never that the change is correct.

Do not connect an automated chain to a control system, and do not give it write access to production postprocessors.

Rules worth writing into the instruction file

Put the requirements that always apply where the assistant always reads them — the workspace instruction file, or the skills. The ones that pay off in practice:

- read the actual project data before proposing a change, and cite the values used;
- read the current source of an item before editing it;
- compile, then run, then compare with the baseline — before reporting anything as done;
- keep the change minimal and explain what it does not cover;
- never send a program to a machine, and never change machine configuration.

CLData

CLData is the intermediate data that the CAM system passes to a postprocessor: an ordered list of technology commands (tool movements, spindle and coolant control, tool changes, canned cycles, auxiliary functions, etc.), each carrying a set of parameters. A postprocessor reads CLData and generates the NC program for a particular machine and control system.

Both postprocessing subsystems — the Postprocessors generator (sppx postprocessors) and the .NET postprocessors — work with the same CLData. The module that reads and writes CLData is shared between them; the low-level interaction with native code is hidden, and each subsystem exposes the data through convenient wrappers. Only the access syntax differs between the two subsystems — the commands and their parameters are identical.

This section documents the CLData commands once, for both subsystems. For every command the parameters are described in a syntax-neutral way first, followed by separate paragraphs showing how to read them from sppx and from .NET. Use the access form of your own subsystem and do not mix the two syntaxes.

What CLData contains

CLData is organized as a small hierarchy - a project, its files, the commands inside each file, and the parameters of each command:

- Project - the whole CAM project. Apart from its operations it carries project-level information - the machine schema, the parts and workpieces, the units, the setup stages, and so on.
- Files - the project's CLData is split into files. There is one file for each CAM operation, an optional file for each subprogram, and one project-level file that holds the project information and contains no commands.
- Commands - each file is an ordered sequence of technology commands: tool movements, spindle and coolant control, tool changes, canned cycles, section markers, auxiliary functions, and so on. The order is the order in which the operation is performed.
- Parameters - every command carries a set of parameters that describe it: the coordinates of a move, the spindle speed, the tool number, a cycle's depths and offsets, and so on. Each command type has its own set of parameters.

A postprocessor walks this hierarchy - file by file, command by command - and for each command reads the parameters it needs to build the corresponding NC block. The next section shows a real listing of such a project; the sections after it explain how to reach the files, commands and parameters from postprocessor code.

The structure of a CLData listing

A project's CLData is not one flat list of commands but a set of files, and inside each file the commands are grouped into nested sections. Knowing this layout makes the command list readable and lets a handler react to the toolpath structure. The example below is a real listing of a simple mill-turn project, produced by a small postprocessor that walks every file and command (the same file-walking pattern as the example in CLData files).

Files

- A project's CLData is a list of files, numbered from `0`. File numbering is 0-based in both subsystems. The numbering of commands *inside* a file, however, differs: it is 1-based in sppx (`for c = 1 to CldFile[f].CmdCount`) and 0-based in .NET (`for (int c = 0; c < file.CmdCount; c++)`).
- File **`0`** is always the project-level file (type `CAMProject`). It carries no commands - it holds the project information, read with the Project operator.
- Every CAM operation becomes one file of type `TechOperation`, in the order of the operations list.
- Files of type `NCSub` are optional - they exist only when the project uses subprograms or cycles-with-subprograms. Unlike operation files, they are not translated automatically; a postprocessor emits them by calling the NC-subroutine operators explicitly (see Operators of work with NC-subroutines).

```
File 0   CAMProject      - project-level data, no commands
File 1   NCSub           - optional: subprogram / cycle bodies
File 2   NCSub
File 3   TechOperation   - one file per CAM operation, in operations-list order
File 4   TechOperation
 ...
File 15  TechOperation
```

Sections inside an operation

- Each operation file is wrapped by `PPFUN TECHINFO(58)` ⋯ `PPFUN ENDTECHINFO(59)` - the operation start / end, carrying the operation's technological data.
- Commands are grouped into nested sections with the `STRUCTURE` command (`ON(71)` opens a section, `OFF(72)` closes it). The three top-level sections are Header, Body and Tail; Header contains an Approach sub-section and Tail a Return sub-section. `STRUCTURE` is also used to group the cutting moves by region / tool pass - for a milling operation, for example, the Body is split into depth Level groups and each level into Offset passes (see `File 15` below). In the command list these nested sections appear as indentation.
- The Header carries the set-up commands, typically in this order:
  - the first `COMMENT` - the CAM operation name;
  - `ORIGIN` - the base coordinate system this operation's toolpath is output in;
  - `LOADTL` - the tool used in the operation;
  - the `COMMENT` right after the tool, starting with `@` - the tool name;
  - `CUTCOM` - activates the tool compensation;

    - PLANE - activates the working plane;
    - SPINDL - starts the spindle;
    - FROM - the tool's initial position; it performs no motion, it only states where the tool already is when the operation starts.
- The set of movement commands depends on the machine kind: GOTO / CIRCLE for 2D/3D mill-turn toolpaths (turning and drilling cycles additionally emit EXTCYCLE and MULTIGOTO for positioning, as in the example below), MULTIGOTO / MULTIARC for 5-axis machines and robots, and EDMMOVE for wire-EDM. The rest of the structure stays largely the same.
- Key points of the toolpath are marked with tags - a PPRINT whose text starts with # (for example `#KeyPoint: StartCutting`, `#Approach:…`, `#Return:…`). A postprocessor reacts to a tag or skips it so the marker does not reach the NC program; see Toolpath tags. These lines are marked `! toolpath tag` in the listing below.

The listing below walks the whole project - the project file, the subprogram files and the operation files - with long parameter lists trimmed to … and the repeating passes collapsed; indentation shows the STRUCTURE nesting (per-command indices are omitted):

```
File 0 — CAMProject   (project level; carries no commands)

File 1 — NCSub   (a subprogram / cycle body; output only when the postprocessor
calls it explicitly)
COMMENT    "Geometry"
PPFUN      STARTSUB(50), 1
GOTO       X 67.8, Y 0, Z 0
PPRINT     "#KeyPoint: StartCutting"                    ! toolpath tag
FEDRAT     F 0.5, MMPR(316), K 0
GOTO       X 30.703, Y 0, Z 0
GOTO       X 33.362, Y 0, Z 2.659
PPFUN      ENDSUB(51), 1

File 2 — NCSub   (another subprogram body; same shape as File 1 — omitted)

File 3 — TechOperation "Lathe facing"   (turning)
STRUCTURE  ON(71)  TYPE "HeaderSection", Comment "$Header$"
  PPFUN      TECHINFO(58), 250, 30.703, 0, 0, 979.484, …    ! the operation starts
  STRUCTURE  ON(71)  TYPE "String", Comment "Header"
    COMMENT    "Lathe facing"                               ! operation name
    ORIGIN     X 0, Y 0, Z 265, PPFUN 0, N 54, …            ! base coordinate
system (G54)
    LOADTL     N 2, ToolChanged 1, X 15, Y 125, …           ! tool
    COMMENT    "@IC16 Re0.2 R OD cutting tool"              ! tool name (starts
with @)
    CUTCOM     ON(71), LENGTH(9)2, …, LEFT(8)               ! tool compensation
    FROM       COUNT 10, MACHINE, X 100, Y 0, Z 500, …      ! initial tool position
```

```
(no motion)
  STRUCTURE  OFF(72) TYPE "String", Comment "Header"
  STRUCTURE  ON(71)  TYPE "String", Comment "Approach"
    STRUCTURE  ON(71)  TYPE "Approach", Comment "Approach"
      PPRINT     "#Approach:[Rule 1]=C; Z10; X; Z"      ! toolpath tag
      SELWORKPIECE"Table"
      SPINDL     ON(71), NO -1000, K 0, MODE CSS(2), SPEED -150   ! spindle
      RAPID      N 10000
      MULTIGOTO  COUNT 1, MACHINE, Z 10, …
      MULTIGOTO  COUNT 1, MACHINE, X 67.8, …
      MULTIGOTO  COUNT 1, MACHINE, Z 2.659, …
    STRUCTURE  OFF(72) TYPE "Approach", Comment "Approach"
  STRUCTURE  OFF(72) TYPE "String", Comment "Approach"
STRUCTURE  OFF(72) TYPE "HeaderSection", Comment "$Header$"
STRUCTURE  ON(71)  TYPE "BodySection", Comment "$Body$"
  GOTO       X 67.8, Y 0, Z 2.659
  COOLNT     ON(71), N 1, #1
  FEDRAT     F 0.5, MMPR(316), K 0
  EXTCYCLE   ON(71)   LATHEFINISH(400), 1, 0, 3, 0, 0.3, …
  EXTCYCLE   CALL(52) LATHEFINISH(400), 1, 0, 3, 0, 0.3, …
  EXTCYCLE   OFF(72)  LATHEFINISH(400), 1, 0, 3, 0, 0.3, …
STRUCTURE  OFF(72) TYPE "BodySection", Comment "$Body$"
STRUCTURE  ON(71)  TYPE "TailSection", Comment "$Tail$"
  STRUCTURE  ON(71)  TYPE "String", Comment "Return"
    STRUCTURE  ON(71)  TYPE "Return", Comment "Return"
      PPRINT     "#Return:[Rule 1]=Z10; X; Z; C"        ! toolpath tag
      RAPID      N 10000
      MULTIGOTO  COUNT 1, MACHINE, Z 10, …
    STRUCTURE  OFF(72) TYPE "Return", Comment "Return"
  STRUCTURE  OFF(72) TYPE "String", Comment "Return"
  PPFUN      ENDTECHINFO(59), 250, 30.703, 0, 0, 979.484, … ! the operation ends
STRUCTURE  OFF(72) TYPE "TailSection", Comment "$Tail$"

…  (Files 4…14 — the other operations, each the same Header / Body / Tail shape
— omitted)

File 15 — TechOperation "2D contouring"   (milling)
STRUCTURE  ON(71)  TYPE "HeaderSection", Comment "$Header$"
  PPFUN      TECHINFO(58), 250, -47.996, -289.976, -16, …  ! the operation starts
  STRUCTURE  ON(71)  TYPE "String", Comment "Header"
    COMMENT    "2D contouring"                            ! operation name
    ORIGIN     X 0, Y 0, Z 274, PPFUN 0, N 55, …          ! base coordinate
system (G55)
    LOADTL     N 1, ToolChanged 1, …, D 8, L 40, …        ! tool
    COMMENT    "@Cylindrical mill L40, D8"                ! tool name (starts
with @)
```

```
    PLANE      XY                                          ! working plane
    SPINDL     ON(71), NO 10544.01, K 0, MODE RPM(0)       ! spindle
    CUTCOM     ON(71), LENGTH(9)1, …, LEFT(8)              ! tool compensation
    FROM       COUNT 10, MACHINE, X 100, Y 0, Z 500, …     ! initial tool position
(no motion)
  STRUCTURE  OFF(72) TYPE "String", Comment "Header"
  STRUCTURE  ON(71)  TYPE "String", Comment "Approach"
    STRUCTURE  ON(71)  TYPE "Approach", Comment "Approach"
      PPRINT     "#Approach:[Rule 1]=C; Z10; X; Z"      ! toolpath tag
      RAPID      N 10000
      AXESBRAKE  COUNT 1, AxisCPos(3) OFF(72)
      MULTIGOTO  COUNT 1, MACHINE, C 90, …
      MULTIGOTO  COUNT 1, MACHINE, Z 10, …
      MULTIGOTO  COUNT 1, MACHINE, X -28.715, …
      MULTIGOTO  COUNT 1, MACHINE, Z 0, …
      MULTIGOTO  COUNT 1, MACHINE, Y -1.804, …
    STRUCTURE  OFF(72) TYPE "Approach", Comment "Approach"
  STRUCTURE  OFF(72) TYPE "String", Comment "Approach"
STRUCTURE  OFF(72) TYPE "HeaderSection", Comment "$Header$"
STRUCTURE  ON(71)  TYPE "BodySection", Comment "$Body$"
  STRUCTURE  ON(71)  TYPE "String", Comment "Level: -4"      ! first depth level
    STRUCTURE  ON(71)  TYPE "String", Comment "Offset 2"     ! first offset pass at
this level
      RAPID      N 10000
      GOTO       X -28.715, Y -1.804, Z 0
      GOTO       X -28.715, Y -1.804, Z -3
      AXESBRAKE  COUNT 1, AxisCPos(3) ON(71)
      COOLNT     ON(71), N 1, #1
      FEDRAT     F 100, MMPM(315), K 256
      GOTO       X -28.715, Y -1.804, Z -4
      FEDRAT     F 100, MMPM(315), K 4
      GOTO       X -28.716, Y -3.004, Z -4
      GOTO       X -32.716, Y -3.001, Z -4
      FEDRAT     F 200, MMPM(315), K 0
      GOTO       X -39.887, Y -2.994, Z -4
      CIRCLE     XC -0.086, YC -0.024, ZC -4, R -39.911, XE -39.882, YE 3, ZE -4
      GOTO       X -32.725, Y 3.001, Z -4
      …
    STRUCTURE  OFF(72) TYPE "String", Comment "Offset 2"
    STRUCTURE  ON(71)  TYPE "String", Comment "Offset 1"     ! next offset pass —
same shape — omitted
      …
    STRUCTURE  OFF(72) TYPE "String", Comment "Offset 1"
    STRUCTURE  ON(71)  TYPE "String", Comment "Offset 0"
      …
    STRUCTURE  OFF(72) TYPE "String", Comment "Offset 0"
```

```
  STRUCTURE  OFF(72) TYPE "String", Comment "Level: -4"
  …  (Levels -8, -12, -16 — the same offset passes at each depth; then the whole set
repeats after
      each C-axis rotation MULTIGOTO C 180 / 270 / 360 — a lot of repeating
passes, omitted)
STRUCTURE  OFF(72) TYPE "BodySection", Comment "$Body$"
STRUCTURE  ON(71)  TYPE "TailSection", Comment "$Tail$"
  COOLNT     OFF(72), N 1, #1
  STRUCTURE  ON(71)  TYPE "String", Comment "Return"
    STRUCTURE  ON(71)  TYPE "Return", Comment "Return"
      PPRINT     "#Return:[Rule 1]=Z10; X; Z; C"        ! toolpath tag
      RAPID      N 10000
      MULTIGOTO  COUNT 1, MACHINE, Z 10, …
      GOHOME     COUNT 1, MACHINE, X 100, …
      GOHOME     COUNT 1, MACHINE, Z 500, …
      GOHOME     COUNT 1, MACHINE, Y 0, …
    STRUCTURE  OFF(72) TYPE "Return", Comment "Return"
  STRUCTURE  OFF(72) TYPE "String", Comment "Return"
  CUTCOM     OFF(72), LENGTH(9)0, …                          ! compensation off
  SPINDL     OFF(72), NO 0, K 0                              ! spindle off
  PPFUN      ENDTECHINFO(59), 250, -47.996, -289.976, …      ! the operation ends
STRUCTURE  OFF(72) TYPE "TailSection", Comment "$Tail$"
```

Access model

A command's parameters come in two flavours:

- Numeric parameters — a flat array of real numbers, addressed by position. This is the original, legacy form.
- Named / structured parameters — typed values addressed by name, possibly nested (objects, arrays). This is the modern form used by newer commands (and available for most parameters of the older ones). The naming model is the same one described in XML properties — access from code.

The table summarizes the ways to read them. Details and examples are given per command in the command reference; the access pattern is always the same.

Access form	sppx	.NET	Notes
Numeric array, by index	`cld[i]`	`cld[i]`	Common to both. The array is filled with the current command's parameters before the command is processed.

Access form	sppx	.NET	Notes
Numeric element, by short name	`cld.X`, `cld.N` …	—	sppx only. A flat, duplicate accessor for some numeric elements. Not available in .NET.
Named / structured, by name	`cmd.Str["Name"]`, `cmd.Flt["Name"]`, `cmd.Int["Name"]`, `cmd.Ptr["Name"]`	`cmd.Str["Name"]`, `cmd.Flt["Name"]`, `cmd.Int["Name"]`, `cmd.Ptr["Name"]`	Same grammar in both (dotted paths, array keys `Name(Key)`, pointer chaining). See XML properties — access from code.
By stable parameter code	`CmdPrm.Flt[code]`	`cmdPrm.Flt[code]`	Used by the extended / probing cycles, where parameters are addressed by a fixed code so their order can change without breaking postprocessors.
Strongly-typed command wrapper	—	`cmd.<Property>` (e.g. `cmd.R`, `cmd.IsCSS`)	.NET only. Each command handler receives a typed `cmd` object with named, documented properties. The modern .NET way; see the API reference.

sppx

In sppx the current command's numeric parameters are placed in the predefined `cld` array before the command's program runs, so `cld[1]`, `cld[2]`, … address them by position; some elements also have short member names such as `cld.X`. Structured parameters are read by name with the `cmd` operator (`cmd.Str["…"]`, `cmd.Flt["…"]`, `cmd.Ptr["…"]`), and the extended cycles use code-based access `CmdPrm.Flt[code]`. These accessors are described in CLData access functions and operators.

.NET

In .NET a postprocessor implements a typed handler per command, for example `OnSpindle(ICLDSpindleCommand cmd, CLDArray cld)`. Inside the handler you can:

- use the strongly-typed `cmd` object — named, documented properties such as `cmd.IsCSS`, `cmd.RPMValue`, `cmd.R`, `cmd.Center` (this is the recommended, modern form);
- read named / structured parameters with `cmd.Str["Name"]`, `cmd.Flt["Name"]`, etc.;
- read the raw numeric array `cld[i]` (the same indices as in sppx);
- use code-based access `cmdPrm.Flt[code]` for the extended cycles.

The typed command interfaces (`ICLD…Command`) and all their properties are described in the .NET SDK API reference — see .NET references.

Locating commands (browsing CLData)

Besides the current command, a postprocessor can browse other commands of the file (look ahead/back, find the next command of a given type, read project / operation context).

- sppx: `CurCode`, `NextCode`, `CodeOfCmd`, `GetCld`, `FindCld` / `GFindCld`, the `CLDFile` operator and the `Project` operator.
- .NET: `CurrentFile.Cmd[i]`, `CurrentFile.FindCommand(...)`, `CurrentFile.IndexOfCmd(...)`, `CurrentCmd.Next` / `Prev` / `NextMotion` / `PrevMotion`, `CurrentOperation.*`, `CLDProject.*`.

Both forms are documented side by side in CLData access functions and operators.

Command reference

Each command is documented once for both subsystems (common parameters → access from sppx → access from .NET). The full list is in Technology commands description.

Project information in CLData

Static project-level data embedded in CLData (machine schema, parts, workpieces, setup stages, WCS, units, etc.) is described in Project information in CLData.

In this section

- CLData access functions and operators - reading CLData from postprocessor code
- Project information in CLData - static project-level data embedded in CLData
- Technology commands description - the list of all CLData commands

CLData access functions and operators

The postprocessor generator's programming language implements several functions, variables and operators that provide information about the contents of CLData. The idea is the same in sppx and in .NET, but the concrete primitives differ, so each topic below documents the sppx form and the .NET form side by side - the parameters of the current command, the other commands of the file, and the project-level data - and points out where one subsystem has no direct counterpart.

For the overall picture of how a postprocessor reaches CLData (the command model, the current command, the `cld` array) see the CLData access model.

In this section

- Named CLData parameters - the common named-parameter model (`Str` / `Int` / `Flt` / `Ptr` access by a dotted/bracketed name); shared by the command, file and project accessors.
- CLD predefined array - the numeric parameter array of the current command (`CLD[i]` / `CLD.Name` in sppx, `CLDArray` in .NET).
- Current command (Cmd operator) - reading the code, name, data and named parameters of the current command, including parameters addressed by a unique code.
- Browsing CLData commands - looking at other commands of the file (look ahead/back, find the next command of a given type, command codes).
- CLData files (CLDFile operator) - access to any command of any loaded CLData file.
- Project information (Project operator) - access to the CAM project-level data embedded in CLData.
- GMA array - the sppx convenience array for processing multi-axis MULTIGOTO movements.

See also

- CLData access model
- Technology commands description
- Project information in CLData

Named CLData parameters

From the CAM system to the postprocessor comes a set of data, called CLData, on the basis of which the postprocessor generates a G-code. The data inside CLData can describe a lot of different parameters that relate to completely different complex objects: technological operation, toolpath, tool, project, workpiece, etc. Therefore, within CLData, they are definitely structured and in fact are not a simple list, a parameter tree. These parameters are accessed from the postprocessor code by a string name. Since we have a parameter tree, then usually the full name of the parameter is a complex string consisting of several identifiers separated by dots and sometimes brackets, i.e.

GroupName.SubGroupName.ParameterName.SubParameterName etc. The level of nesting of parameters can be different - from one to several identifiers separated by dots. An example of the real name of the parameter responsible for the smoothing radius of the toolpath inside the operation: "Operation.Smoothing.Radius".

Access from sppx

The syntax for accessing a parameter from the postprocessor program code is as follows.

```
ObjectKeyWord.TypeKeyWord["ParameterName"]
```

Here ObjectKeyWord is a keyword that tells which object from CLData we are requesting parameters. Now, as an ObjectKeyWord can be:

- Cmd - current CLData command.
- CLDFile[*index1*].Cmd[*index2*]. CLData command with index index2 from the cld-file with index index1. Here index1 and index2 are any integer numbers.
- Project- CAM system project, in which the parameters relating to all operations are grouped, and not to an individual operation, for example, the workpiece.

TypeKeyWord is a keyword that determines the form (string, number, etc.) to get the value of the parameter. Parameters can be of different types: string, integer, floating-point numbers, complex (parameters of this type have no eigenvalue, but they can have child properties of any of the types described here; required to combine several parameters into a group), array (as complex ones do not have their own values and can have child properties, however, the child properties of arrays, as a rule, have a common type and, therefore, the same name, so the access to the child properties of arrays is carried out not by name, but by index).

The following keywords are available.

- `Str["ParameterName"]` - returns the value of the command parameter with the name ParameterName as a text String.

- `Int["ParameterName"]` - returns the value of the command parameter with the name ParameterName as an Integer number.
- `Flt["ParameterName"]` - returns the value of the command parameter with the name ParameterName as a Real number (floating point number).
- `Ptr["ParameterName"]` - returns a reference (not value) to the command parameter with the ParameterName name. It is convenient to use for access to parameters having type complex and array. If a parameter with that name in this command does not exist, then the operator returns the value 0, otherwise the number is nonzero. Thus, the test for equality to zero can be used to check the existence of a parameter.

```
if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
  C = cmd.Flt["Axes(AxisCPos).Value"]
end
```

From the obtained reference through the point may be accessed either to the value of this parameter or to one of the child parameters. To do this, after the point again need to specify one of the commands described here: Str, Int, Flt, Ptr, Item, ItemCount, Name. If the following instructions do not specify the parameter name, it means that it refers to this parameter. For example, the instruction `Cmd.Ptr["Parameter1"].Int` means take the value of the parameter with the name `Parameter1` as an integer number. A record `Cmd.Ptr["Parameter1"].Int["SubParameter1"]` means to take an integer value of the child parameter `SubParameter1` inside parameter `Parameter1`.

`Cmd.Item[Index]` - returns a reference to the child parameter by index. Parameter's indexing starts with 1.

`Cmd.ItemCount` - returns the number of child parameters in list.

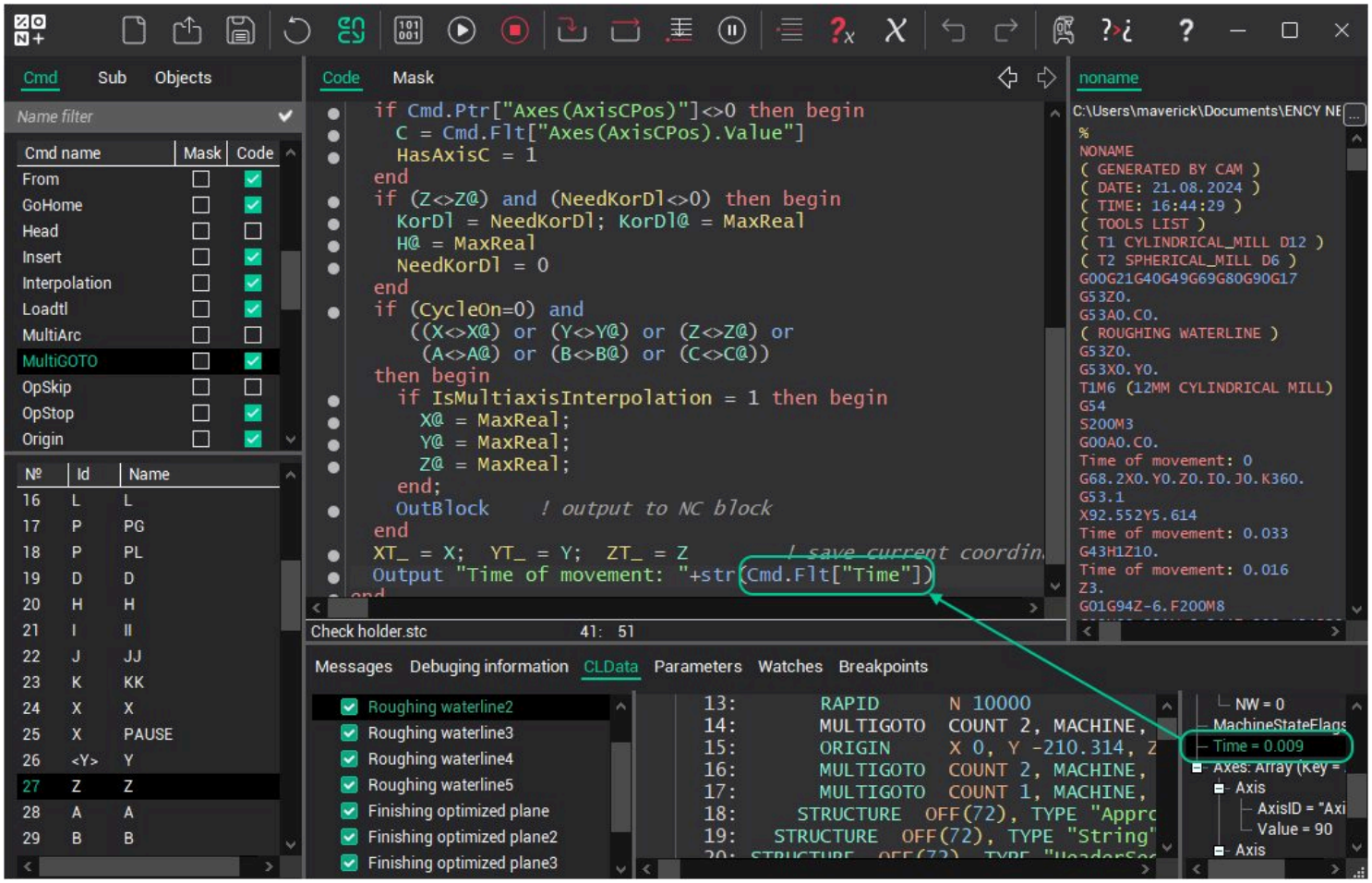
For array type parameters, all child parameters one level down can have the same name. In this case, access by name is not possible; instead, you must use access by index.

- `ObjectKeyWord.Ptr["NameOfArrayParameter"].Item[Index]` - returns a reference to the child parameter by index. Parameter's indexing starts with 1.
- `ObjectKeyWord.Ptr["NameOfArrayParameter"].ItemCount` - returns the number of child parameters in list.

In addition to index access, access to the child array parameters in some cases can be implemented by a key. If the array has a defined Key parameter, which specifies the name of the key field in the child parameters, to obtain access to a particular element of the array after the array name must specify the key value in parentheses. For example, the instruction `Cmd.Ptr["ArrayName(KeyValue)"]` means to take the parameter ArrayName from the current command, where to find the child property with a key field equal KeyValue.

As the parameter name in them can be not only a string constant, as shown here, but the string variables and expressions.

There is a convenient way to form a string to access the desired parameter. All parameters for CLData objects are shown in a separate tree on the CLData tab in its right side. It is necessary to select the necessary command in the list, or the project in the list of operations. At the same time in the tree on the right you will see the parameters of the selected object. Then you need to find the desired parameter in this tree and double-click on it. The line of access to the specified parameter will be inserted into the current cursor position in the code editing window.



Example of MultiGoto command processing using various methods to access elements of array parameter Axes (by index, by key, and by compound name):

```
program MultiGoto
  i: Integer
  AxisName: String
  for i = 1 to Cmd.Ptr["Axes"].ItemCount do begin
    AxisName = cmd.Ptr["Axes"].Item[i].Str["AxisID"]
    if AxisName="AxisXPos" then begin
      X = cmd.Ptr["Axes"].Item[i].Flt["Value"]
    end else
```

```
    if AxisName="AxisYPos" then begin
      Y = cmd.Ptr["Axes("+Str(i)+")"].Flt["Value"]
    end else
    if AxisName="AxisZPos" then begin
      Z = cmd.Flt["Axes("+Str(i)+").Value"]
    end else
    if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
      C = cmd.Flt["Axes(AxisCPos).Value"]
    end
  end
  OutBlock
end
```

More examples of named-parameter access are given in Current command.

Access from .NET

.NET uses the same named-parameter model, exposed by the `INamedProperty` interface. Every CLData object that carries named parameters implements it - the current command (`ICLDCommand`), an arbitrary command taken through CLDFile, and the project (`ICLDProject`). The keywords match the sppx ones one-to-one:

sppx	.NET	Description
`Str["Name"]`	`Str["Name"]`	the parameter value as a string
`Int["Name"]`	`Int["Name"]`	the parameter value as an integer
`Flt["Name"]`	`Flt["Name"]`	the parameter value as a floating-point number
-	`Bol["Name"]`	the parameter value as a boolean
`Ptr["Name"]`	`Ptr["Name"]`	a reference to the (complex) parameter as an `INamedProperty`
-	`Arr["Name"]`	a reference to an array parameter as an `INamedPropertiesCollection`

sppx	.NET	Description
`Item[i]`	`Arr["Name"][i]`	the i-th child of an array parameter (0-based in .NET)
`ItemCount`	`Arr["Name"].TopItem`	the maximal index in an array parameter
`Name`	`Name`	the parameter's own name

> Index base. When the children of an array parameter are addressed by index, the numbering differs between the two subsystems: it is 1-based in sppx (`Item[1]` ... `Item[ItemCount]`) and 0-based in .NET (`Arr["Name"][0]` ... `Arr["Name"][TopItem]`). Correspondingly, the sppx `ItemCount` is the number of children, whereas the .NET `TopItem` is the maximal index - one less than the count - so a .NET loop runs `for (int i = 0; i <= coll.TopItem; i++)`. (This concerns the named array-parameter children only; the numeric CLD array is 1-based in both subsystems.)

As in sppx, the name can be a compound string with dots and brackets - `Ptr["StartPoint.X"]`, `Ptr["Axes(AxisCPos).Value"]` - resolved to any nesting depth; the character case does not matter. In addition, `INamedProperty` offers `FindChild(name)`, `GetNextChild(child)`, `Parent`, `SimpleType` (the `NamedParamType` of the value) and the typed value accessors `ValueAsString` / `ValueAsInteger` / `ValueAsDouble` / `ValueAsBoolean`.

```
public override void OnMultiGoto(ICLDMultiGotoCommand cmd, CLDArray cld)
{
    var axes = cmd.Arr["Axes"];
    for (int i = 0; i <= axes.TopItem; i++) {
        string axisName = axes[i].Str["AxisID"];
        double value = axes[i].Flt["Value"];
        // ... output the axis
    }
    // or address a single axis directly by its key:
    if (cmd.Ptr["Axes(AxisCPos)"] != null)
        C = cmd.Flt["Axes(AxisCPos).Value"];
}
```

See also

- Current command (Cmd operator)
- CLData files (CLDFile operator)
- Project information (Project operator)
- CLData access functions and operators

CLD predefined array

The CLD array holds the numerical parameters of the currently executing CLData command. Before a command handler is called, the command's parameters are placed into this array; inside the handler the parameters are analyzed by reading individual array elements, and on the basis of that analysis blocks of the NC-program are generated.

Command parameters are accessed either by indexing the appropriate item in the array or by a unique parameter name.

Access from sppx

The array CLD is an array of real numbers. It is designed to store numerical parameters of currently executing technological CLData command. Before calling the handler of the current command its parameters are filled in this array. Inside the handler command parameters can be analyzed by accessing individual array elements and on the basis of analysis can be generated blocks of the NC-program.

Command parameters are accessed either by indexing the appropriate item in the CLD array or by unique parameter name.

To access by index use the common array syntax: `CLD[i]` - i-th array item - real number, i - item index - positive integer. Index of array element can be a variable or an expression of integer type. For example: `CLD[3]`, `CLD[n]`, `CLD[2*n+1]` etc. The total number of elements in the array CLD contained in the predefined variable RecNum.

To access parameters by name use the following syntax: `CLD.Parameter name`, where Parameter name - unique identifier of the parameter. For example, `CLD.X`, `CLD.Mode` and so on. Parameter names are specific for each command. The detailed description of CLData technology commands and their parameters is found in the technology commands description. Also you can see the list of parameters at the Current parameters panel of the Mask page of the main window and on the CLData tab at the bottom of the main window.

Example of using CLD array:

```
program Circle
  if CLD.R > 0 then INTERP_ = 3
  else INTERP_ = 2 ! G3/G2
  X = cld[5]
  Y = cld[6]
  Z = cld[7]
  I = CLD.Xc
  J = CLD.Yc
  R = abs(cld.R)
```

```
  OutBlock
end
```

The above CIRCLE command handler outputs into NC-code blocks of moving in a circle G2 or G3 (clockwise or counterclockwise) with the endpoint (X, Y, Z), center (I, J) and radius (R).

Access to the parameters of CLData command, including and not numeric, can also be obtained by the operator Cmd.

Access from .NET

A command handler receives the same numeric parameter array as its second argument - an instance of the `CLDArray` class (conventionally named `cld`). Its members mirror the sppx CLD array:

sppx	.NET	Description
`CLD[i]`	`cld[i]`	the i-th parameter, a real number; the index is 1-based ([1..Count]), the same as in sppx
`CLD.Name`	`cld["Name"]`	the parameter addressed by its unique textual name (case-insensitive)
`RecNum`	`cld.Count`	the number of parameters in the array

The same CIRCLE handler in .NET:

```
public override void OnCircle(ICLDCircleCommand cmd, CLDArray cld)
{
    int interp = cld["R"] > 0 ? 3 : 2;   // G3 / G2
    double x = cld[5];
    double y = cld[6];
    double z = cld[7];
    double i = cld["Xc"];
    double j = cld["Yc"];
    double r = Math.Abs(cld["R"]);
    // ... output the block
}
```

In .NET the parameters are usually read from the strongly-typed `cmd` object instead (`cmd.EP.X`, `cmd.R`, ...) - see Current command; the raw `cld` array is available for the cases the typed members do not cover.

See also

- Current command (Cmd operator)
- Named CLData parameters
- CLData access functions and operators

Current command (Cmd operator)

The current command is the CLData command whose handler is being executed. Besides the numeric CLD array, a postprocessor can read the command's code, name, textual data and any of its named parameters (including nested ones). The sppx Cmd operator and the .NET strongly-typed `cmd` object provide this.

Access from sppx

The Cmd operator provides a flexible mechanism for working with the current CLData command. It returns a reference to the current command, on which you can request a code, name and parameters of this command. Format of access to specific command data following.

`Cmd.Code` - returns a unique numeric code of the CLData command. Each command is uniquely identified by this code. List of command codes listed in the technology commands description.

`Cmd.Name` - returns the text name of the command or command parameter.

`Cmd.Data` - for individual CLData commands, which have a single string parameter (such as Comment, PartNo, etc.), returns string value. For the rest of the commands returns a string representation of the command, exactly what is displayed on the tab CLData of main window.

`Cmd.Data[Index]` - an alternative way to access an array of numeric parameters of command CLD. Returns the real value of the parameters array element with the index Index.

The following below are a few language structures allow access to the parameters of CLData command by parameter name. See Named CLData parameters topic for more information. Each CLData command can be characterized by its own set of named parameters. The names of the parameters with description for each command are listed in the technology commands description. The parameter names can be seen on CLData tab at the bottom of the main window.

`Cmd.Str["ParameterName"]` - returns the value of the command parameter with the name ParameterName as a text String.

`Cmd.Int["ParameterName"]` - returns the value of the command parameter with the name ParameterName as an Integer number.

`Cmd.Flt["ParameterName"]` - returns the value of the command parameter with the name ParameterName as a Real number (floating point number).

`Cmd.Ptr["ParameterName"]` - returns a reference to the command parameter with the ParameterName name.

Consider a few examples of CLData command parameters using.

Example 1. Inside the tool loading command handler LoadTl to NC-program output the string with address T, with number of the tool and with the name of the turret head in parentheses.

```
program LoadTl
  Output "T" + Cmd.Int["ToolID"] + "(" + Cmd.Str["RevolverID"] + ")"
end
```

A possible result of the output to the NC-code: T3 (TopTurret)

Example 2. Example of MultiGoto command processing using various methods to access elements of array parameter Axes.

```
program MultiGoto
  i: Integer
  AxisName: String
  for i = 1 to Cmd.Ptr["Axes"].ItemCount do begin
    AxisName = cmd.Ptr["Axes"].Item[i].Str["AxisID"]
    if AxisName="AxisXPos" then begin
      X = cmd.Ptr["Axes"].Item[i].Flt["Value"]
    end else
    if AxisName="AxisYPos" then begin
      Y = cmd.Ptr["Axes("+Str(i)+")"].Flt["Value"]
    end else
    if AxisName="AxisZPos" then begin
      Z = cmd.Flt["Axes("+Str(i)+").Value"]
    end else
    if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
      C = cmd.Flt["Axes(AxisCPos).Value"]
    end
  end
  OutBlock
end
```

Example 3. Example of coordinate system setup command processing Origin. It analyzes the OriginType property's value. If it is zero, output the standard workpiece coordinate system selection command G54-G59. Number of the coordinate system is taken from the parameter CSNumber. Otherwise the local coordinate system transfer command G92 X Y Z is formed. The values of the displacements along the axes of the coordinate system are taken in different ways from the corresponding child properties of the `MCS.OriginPoint` parameter.

```
program Origin
  if Cmd.Int["OriginType"]=0 then begin !G54-G59
    Output "G" + Cmd.Str["CSNumber"]
```

```
  end else begin                          !G92 X Y Z
    X = Cmd.Flt["MCS.OriginPoint.X"]
    Y = Cmd.Ptr["MCS.OriginPoint.Y"].Flt
    Z = Cmd.Ptr["MCS.OriginPoint"].Flt["Z"]
    Output "G92 X" + Str(X) + " Y" + Str(Y) + " Z" + Str(Z)
  end
end
```

Cmd operator (parameters with a unique code)

For probing cycles, parameters can be written to the CLD array in a different order. Each parameter has a unique code by which you can get the value of the parameter.

The syntax for accessing the parameters of the CLData command by numerical code:

- `CmdPrm.ItemCount` - number of parameters for the command;
- `CmdPrm.Flt[Code]` - return the value of the parameter by its code in the Double value;
- `CmdPrm.Int[Code]` - return the value of the parameter by its code in the Integer value;
- `CmdPrm.Str[Code]` - return the value of the parameter by its code in the String value;
- `CmdPrm.Bol[Code]` - return the value of the parameter by its code in the Boolean value;
- `CmdPrm.IndexOfCode(Code)` - find parameter index by its code;
- `CmdPrm.Item[i].ItemType` - parameter type by index (String, Integer, Double, Boolean);
- `CmdPrm.Item[i].Flt` - parameter value by index in the Double value;
- `CmdPrm.Item[i].Int` - parameter value by index in the Integer value;
- `CmdPrm.Item[i].Str` - parameter value by index in the String value;
- `CmdPrm.Item[i].Bol` - parameter value by index in the Boolean value;
- `CmdPrm.Item[i].Code` - numeric parameter code by index.

Access from .NET

In .NET the current command is the first argument of the handler - a strongly-typed interface (`ICLDCircleCommand`, `ICLDGotoCommand`, ...) derived from `ICLDCommand`. Reading the typed properties (`cmd.EP.X`, `cmd.R`, ...) is the recommended way; the members below cover the same information as the sppx Cmd operator:

sppx	.NET (**`ICLDCommand`**)	Description
`Cmd.Code`	`cmd.CmdTypeCode`	the unique numeric type code of the command
-	`cmd.CmdType`	the command type as the `CLDCmdType` enumeration

sppx	.NET (`ICLDCommand`)	Description
`Cmd.Name`	`cmd.Caption`	the command name as it appears in windows
`Cmd.Data`	`cmd.CLDataS`	the textual data of the command (or its string representation)
`Cmd.Data[i]`	`cmd.CLD[i]`	the i-th numeric parameter (see CLD array)
-	`cmd.Index`	the index of the command in its CLData file

The named-parameter access (`Str` / `Int` / `Flt` / `Ptr`) is identical, because `ICLDCommand` implements the common named-property model:

```
public override void OnLoadTl(ICLDLoadTlCommand cmd, CLDArray cld)
{
    nc.OutWithN("T" + cmd.Int["ToolID"] + "(" + cmd.Str["RevolverID"] + ")");
}
```

Nested and array parameters use `cmd.Ptr["Name"]`, `.Arr["Name"]`, `.Item[i]` and the compound `"Group.Sub.Param"` / `"Array(Key)"` name syntax, exactly as in sppx - see Named CLData parameters.

Parameters with a unique code. In .NET postprocessors, everything is almost the same, but access is via the `cmd.Prm` keyword (`cmd.Prm.Flt[code]`, `cmd.Prm.Int[code]`, `cmd.Prm.IndexOfCode(code)`, `cmd.Prm.Item[i]...`).

See also

- Named CLData parameters
- CLD predefined array
- Browsing CLData commands
- CLData access functions and operators

Browsing CLData commands

Besides the current command, a postprocessor can look at other commands of the file - read the code of the current or the following command, look ahead for a command of a given type, or walk the whole file. sppx offers a set of dedicated functions for this; .NET does the same through the navigation members of `ICLDCommand` and the CLDFile object. Command codes/types are the common thread: sppx identifies a command type by a numeric code (obtained from a name with `CodeOfCmd`), .NET by the `CLDCmdType` enumeration.

Access from sppx

CurCode function

CurCode function returns the code of the current CLData command. Each CLData command has a unique numerical identifier - code. CLData command codes are listed in the technology commands description. Example of a function inside the command handler CIRCLE below.

```
program Circle
  Print CurCode
end
```

As a result of the program in debug window output the number 15000.

The code of a following CLData command can be obtained with NextCode function. For the code of arbitrary commands on its textual name should use the CodeOfCmd function.

NextCode function

The NextCode function returns the code of technological command, which is located immediately after the current command. Each CLData command has a unique numerical identifier - code. CLData command codes are listed in the technology commands description. The code of a current CLData command can be obtained with CurCode function. For the code of arbitrary commands on its textual name should use the CodeOfCmd function.

```
if NextCode=CodeOfCmd("Fini") then
Output "M02"
```

CodeOfCmd function

`CodeOfCmd(CommandName: String)` returns the code of CLData command, which name is passed as a parameter. Each CLData command has a unique numerical identifier - code. CLData command codes are listed in the technology commands description. The function has the following format.

```
CodeOfCmd(CommandName: String): Integer
```

CommandName - text name of the command. It must be one of the command names that can be seen in the CLData list on the left side of the main window. The returned value can be used with operators like CurCode, NextCode, GetCLD, `CLDFile[i].Cmd[j].Code` (see the CLDFile operator).

```
if NextCode=CodeOfCmd("Fini") then
Output "M02"
```

CLDCounter function

CLDCounter function returns the number of the current technology command in the technology command list file. For flexible access to arbitrary CLData command on its index in the file you can use the CLDFile operator.

GetCLD function

Use the GetCLD operator to browse CLData commands in the order they appear in the file. Browsing is possible only in forward direction. Syntax:

```
{CldCode =} GetCLD(i: Integer; a: Array of Real)
```

Here:

- CldCode - GetCLD optionally returns the code of retrieved CLData command.
- i - relative index of the queried CLData command:
  - 0 - current command,
  - 1 - next command,
  - 2 - second command from the current and so on.
- a - a dynamic array of real, that will receive the data of the retrieved CLData command. It must be defined before the operator call.

For example, suppose the CLDATA commands list is as the following and suppose the current command is number 9:

```
5: CUTCOM ON(71),LENGTH(9) 2,X 0,Y 0,Z 0,N 0,K 0,M 0,LEFT(8)
6: RAPID N 10000
7: GOTO.abs X 134.533,Y 99.684,Z 80
8: RAPID N 10000
9: GOTO.abs X 134.533,Y 99.684,Z 74.400
```

```
10: FEDRAT N 50,K 4,MMPM(315)
11: GOTO.abs X 134.533,Y 99.684,Z 73.400
12: FEDRAT N 200,K 0,MMPM(315)
13: PLANE XY(33)
```

The technology command GOTO.abs program is the following:

```
program AbsMov
  a: Array of Real ! The array must be defined beforehand
  c: Integer ! This variable will receive the operation code
  c = GetCld(1, a) ! Retrieving the next command data
  if c = CodeOfCmd("FEDRAT") then
    ! Output the value of the first item of CLD array of
  Print "FEDRAT.CLD[1]=", Str(a[1]) ! the next command
end
```

The result of the program execution will be:

```
FEDRAT.CLD[1]=50.
```

To get access to the parameters of arbitrary technological command can also use the CLDFile operator.

GetCLDStr function

GetCLDStr returns a string representation of current CLData command with it's parameters similar to that displayed in the textual representation CLData window.

For example, inside the command handler AbsMov function can return a string like the following.

```
GOTO.abs X 134.533, Y 99.684, Z 80
```

FindCld function

Use the FindCld function to locate command named CmdName in the current CLData file. StartIndex parameter defines the first scanned command. StartIndex is relative to the first command in the current CLData file. If StartIndex parameter is omitted, then operator will scan commands starting with the command immediately following the current. The returned value is the index of the found command. When no command was found operator returns -1. The Data parameter must be a declared array of real. Data will receive the values from the found command's CLData array. Syntax:

```
N = FindCld({<StartIndex>, }<CmdName>, <Data>)
```

For example, suppose the following CLDATA and suppose the current command number is 0.

```
0: PARTNO "Bottle"
1: PPFUN …..
2: COMMENT "Roughing Waterline"
3: LOADTL N 2,X 0,Y 0,Z 0,D 8….
4: SPINDL ON(71),NO 397.887,K 0,MODE RPM(0)
5: CUTCOM ON(71),LENGTH(9) 2,X 0,Y 0,Z 0,N 0,K 0,M 0,LEFT(8)
6: RAPID N 10000
7: GOTO.abs X 134.533,Y 99.684,Z 80
8: RAPID N 10000
```

PartNo program is the following:

```
program PartNo
  V: Array of Real
  NCircle: Integer
  NGoto: Integer
  NCircle = FindCld("CIRCLE", V)
  NGoto = FindCld("GOTO.abs", V)
  if NGoto > NCircle then
    Print "The first movement is an arc"
  else
    Print "The first movement a cut"
End
```

In result of program execution the message "The first movement a cut" will be print out.

To get access to the parameters of arbitrary technological command can also use the CLDFile operator.

GFindCld function

The GFindCld function is similar to the FindCLD function, but it is looking for a command named CmdName in all of CLData files. StartIndex parameter defines the first scanned command. StartIndex is relative to the first command in the first CLData file. If StartIndex parameter is omitted, then operator will scan commands starting with the command immediately following the current. The returned value is the index of the found command. When no command is found operator returns -1. The Data parameter must be a declared array of real. Data will receive the values from the found command's CLData array. Syntax:

```
N = GFindCld({<StartIndex>, }<CmdName>, <Data>)
```

To get access to the parameters of arbitrary technological command can also use the CLDFile operator.

Access from .NET

.NET does not have the sppx browsing functions; instead each command (`ICLDCommand`) knows its neighbours and can search from itself, and the CLDFile object searches within a file. Command types are given by the `CLDCmdType` enumeration rather than by a numeric code obtained from a name.

sppx	.NET	Description
`CurCode`	`cmd.CmdTypeCode` / `cmd.CmdType`	type code / type of the current command
`CLDCounter`	`cmd.Index`	index of the current command in its file
`CodeOfCmd("NAME")`	`CLDCmdType.Name`	a command type - a named enum value instead of a looked-up code
`GetCLD(0, a)` / `GetCLD(1, a)`	`cmd` / `cmd.Next`	the current / the next command (`cmd.Prev` for the previous)
-	`cmd.NextMotion` / `cmd.PrevMotion`	the next / previous motion command
`NextCode`	`cmd.Next?.CmdType`	type of the following command
`FindCld(name, data)`	`cmd.FindNextCommand(type[, offset[, count]])`	find the next command of a type in the same file
`GFindCld(name, data)`	`cmd.FindNextCommandOfProject(type[, ...])`	find the next command of a type across all files
`GetCLDStr`	`cmd.CLDataS`	the textual representation of the command

`cmd.Next` / `Prev` return `null` past the ends of the file; the `FindNext...` methods return `null` when nothing is found. The `CLData` array of a found command is read through its `CLD` member (see CLD array), so no separate output array argument is needed.

```
// sppx: if NextCode = CodeOfCmd("Fini") then Output "M02"
if (cmd.Next != null && cmd.Next.CmdType == CLDCmdType.Fini)
    nc.OutWithN("M02");

// sppx: NGoto = FindCld("GOTO.abs", V)
```

```
ICLDCommand goto = cmd.FindNextCommand(CLDCmdType.Goto);
if (goto != null) {
    double firstParam = goto.CLD[1];
    // ...
}
```

To reach an arbitrary command by index, or to scan a whole file or project, use the CLDFile / project objects (`FindCommand`, `IndexOfCmd`).

See also

- Current command (Cmd operator)
- CLData files (CLDFile operator)
- CLData access functions and operators

CLData files (CLDFile operator)

A CAM project's CLData is split into files - one per technological operation, plus NC-subroutine files and the project-level file. A postprocessor can reach any command of any file (not only the current one) to look ahead, gather tool lists, and so on. sppx does this with the CLDFile operator; .NET with the `ICLDProject.CLDFiles` list of `ICLDFile` objects.

Access from sppx

The CldFile operator is intended to obtain access to CLData files data from the postprocessor code. It provides information only for reading, ie can only be to the right of the assignment operator. Syntax:

- `CldFile.FileCount` - returns the number of CLData loaded files;
- `CldFile.CurrentFile` - index of the current CLData file (CLData files are indexes starting with number 0);
- `CldFile.FirstEnabledFile` - index of the first CLD-file (operation) that is enabled in the list on the CLData tab;
- `CldFile.LastEnabledFile` - index of the last CLD-file (operation) that is enabled in the list on the CLData tab;
- `CldFile.CurrentCmd` - index of the current CLData command in the current CLData file (1 based indices);
- `CldFile[<FileIndex>].Enabled` - returns 1 when the file number FileIndex is enabled, 0 - file is disabled;
- `CldFile[<FileIndex>].FileType` - returns the type of the CLData file: 0 - technological operation, 1 - NC-subroutine, 2 - CAM project settings;
- `CldFile[<FileIndex>].IsNCSub` - returns 1 when the file is a file of NC-subprogram, 0 - when file is not NC-subprogram;
- `CldFile[<FileIndex>].CmdCount` - number of commands in the CLData file number FileIndex;
- `CldFile[<FileIndex>].Cmd[<CmdIndex>]` - returns a reference to the CLData command numbered CmdIndex in a file with an index FileIndex. After this command through a point is acceptable to specify any of the instructions defined for the operator Cmd. For example: `CldFile[<FileIndex>].Cmd[<CmdIndex>].Code` - code of the CLData command with number CmdIndex;
- `CldFile[<FileIndex>].Cmd[<CmdIndex>].Data` - returns a string data of the CLData command with number CmdIndex (for example, a comment line). If the command does not contain string data, then returns its string representation.
- `CldFile[<FileIndex>].Cmd[<CmdIndex>].Data[<CldIndex>]` - value of CLD array item with number CldIndex.
- `CldFile[<FileIndex>].Cmd[<CmdIndex>].Name` - the CLData command or CLData command parameter name;

- `CldFile[<FileIndex>].Cmd[<CmdIndex>].Str |.Int |.Flt |.Ptr` - value of the command parameter by its name in appropriate form; see Named CLData parameters topic for more information

The example below prints out numbers and comments of all tools used in the program.

```
sub PrintAllTools
  i: Integer
  j: Integer
  for i = 0 to CLDFile.FileCount-1 do begin
    if (CldFile[i].Enabled>0) and (CldFile[i].IsNCSub=0) then begin
      for j = 1 to CldFile[i].CmdCount do begin
        ! If the given command is "LoadTl"
        if CldFile[i].Cmd[j].Code = CodeOfCmd("LoadTl") then begin
          ! Print the turret head identiefer
          Print "RevolverID = " + CldFile[i].Cmd[j].Str["RevolverID"]
          ! Print the tool number
          Print "Tool number = ", CldFile[i].Cmd[j].Data[1]
          ! Print the tool comment
          if CldFile[i].Cmd[j+1].Code = CodeOfCmd("Comment") then
            Print "Tool comment: " + CldFile[i].Cmd[j+1].Data
        end
      end
    end
  end
subend
```

Access from .NET

The list of files is reached from the project object as `CLDProject.CLDFiles` (an `ICLDFileList`); the current file is available directly as `CurrentFile`. Each element is an `ICLDFile`. Note that the file-command index in .NET is 0-based (`[0..CmdCount-1]`), whereas the sppx `CLDFile[f].Cmd[c]` command index is 1-based.

`ICLDFileList`:

sppx	.NET	Description
`CldFile.FileCount`	`CLDProject.CLDFiles.FileCount`	number of loaded files
`CldFile[f]`	`CLDProject.CLDFiles[f]`	the file with index `f`
`CldFile.CurrentFile`	`CurrentFile`	the file being postprocessed

`ICLDFile`:

sppx	.NET	Description
`CldFile[f].CmdCount`	`file.CmdCount`	number of commands in the file
`CldFile[f].Cmd[c]`	`file.Cmd[c]` / `file[c]`	the command with index `c` (0-based)
`CldFile[f].Cmd[c].Code`	`file.GetCommandType(c)`	the type of the command with index `c` (`CLDCmdType`)
`CldFile[f].Enabled`	`file.Enabled`	whether the file is postprocessed
`CldFile[f].FileType`	`file.FileType`	the file type (`CLDFileType.TechOperation` / `NCSub` / `CAMProject`)
`CldFile[f].IsNCSub`	`file.FileType == CLDFileType.NCSub`	whether the file is an NC-subroutine
-	`file.Caption` / `file.ID`	a short description / the unique identifier of the file
-	`file.Index`	the file's index in the project
-	`file.Project`	the project the file belongs to
-	`file.IndexOfCmd(type, from, to)`	index of the first command of a type in a range (`-1` if none)
-	`file.IndexOfCmdName(name, from, to)`	the same, by the type name
-	`file.FindCommand(type, from[, to])`	the first command of a type in a range (`null` if none)

The same "print all tools" scan in .NET:

```
foreach... // over CLDProject.CLDFiles
for (int f = 0; f < CLDProject.CLDFiles.FileCount; f++) {
    ICLDFile file = CLDProject.CLDFiles[f];
    if (!file.Enabled || file.FileType == CLDFileType.NCSub)
```

```
        continue;
    for (int c = 0; c < file.CmdCount; c++) {
        ICLDCommand command = file.Cmd[c];
        if (command.CmdType == CLDCmdType.LoadTl) {
            nc.OutWithN("RevolverID = " + command.Str["RevolverID"]);
            nc.OutWithN("Tool number = " + command.CLD[1]);
            ICLDCommand next = file.Cmd[c + 1];
            if (next != null && next.CmdType == CLDCmdType.Comment)
                nc.OutWithN("Tool comment: " + next.CLDataS);
        }
    }
}
```

See also

- Named CLData parameters
- Current command (Cmd operator)
- Browsing CLData commands
- Project information (Project operator)
- CLData access functions and operators

Project information (Project operator)

Part of CLData is project-level data - information that relates to the whole CAM project rather than to an individual operation (the machine schema, the parts and workpieces, setup stages, and so on). A postprocessor reads it with the sppx Project operator or the .NET `ICLDProject` object. The catalogue of the available project properties and their meaning is described in Project information in CLData.

Access from sppx

The Project operator allows you to get access from the code of the postprocessor to the information about the CAM system project (information that relates to the entire project, and not to individual operations), which lies in the CLData. Operator syntax is as follows.

```
Project.Ptr|Str|Int|Flt["ParameterName"]
```

Here, Project is the key word to access CAM system project level parameters.

Ptr or Str or Int or Flt is a keyword that defines the form in which you want to represent the value of a property - as a string, as an integer, as a floating point number, or as a pointer to a property. Read more about this in Named CLData parameters topic.

Further in square brackets follows the string identifier of the property whose value is to be obtained. It can be a string constant, as shown here, as well as any string expression or variable.

For a description of possible properties and their values, see the appendix in the Project information in CLData article.

Consider an example of obtaining information about the workpiece using this operator.

```
                    PrintAllWorkpiece
sub PrintAllWorkpiece
  i: Integer
  j: Integer
  px, py, pz, pv: Real
  for i = 1 to Project.Ptr["Parts"].ItemCount do begin
    Output "( Part" + str(i) + ".Workpiece )"
    for j = 1 to Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].ItemCount do begin
      Output "(     Primitive " + str(j) + " )"
      case
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Int["PrimitiveType"] of
        0: begin !Unknown
          Output "(         Type: Unknown )"
        end
        1: begin !Empty
```

```
          Output "(         Type: Empty workpiece )"
        end
        2: begin !Faces
          Output "(         Type: Faces )"
          px =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.X"]
          py =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.Y"]
          pz =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.Z"]
          Output "(         Min point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
          px =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.X"]
          py =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.Y"]
          pz =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.Z"]
          Output "(         Max point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
        end
        3: begin !Casting
          Output "(         Type: Casting )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Stock"]
          Output "(         Stock: " + str(pv) + " )"
          px =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.X"]
          py =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.Y"]
          pz =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Min.Z"]
          Output "(         Min point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
          px =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.X"]
          py =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.Y"]
          pz =
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Box.Max.Z"]
          Output "(         Max point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
        end
        4: begin !Box
          Output "(         Type: Box )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Min.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Min.Y"]
```

```
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Min.Z"]
          Output "(         Min point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Max.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Max.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Max.Z"]
          Output "(         Max point: (" + str(px) + ", " + str(py) + ", " +
str(pz) + " ) )"
        end
        5: begin !RevBody
          Output "(         Type: Turn envelope )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Z"]
          Output "(         Origin: (" + str(px) + ", " + str(py) + ", " + str(pz) +
" ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Z"]
          Output "(         Axis: (" + str(px) + ", " + str(py) + ", " + str(pz) + "
) )"
        end
        6: begin !Cylinder
          Output "(         Type: Cylinder )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Z"]
          Output "(         Origin: (" + str(px) + ", " + str(py) + ", " + str(pz) +
" ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Z"]
          Output "(         Axis: (" + str(px) + ", " + str(py) + ", " + str(pz) + "
) )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMin"]
          Output "(         Min level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMax"]
          Output "(         Max level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["ROut"]
          Output "(         Outer radius: " + str(pv) + " )"
        end
        7: begin !Tube
          Output "(         Type: Tube )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Z"]
```

```
          Output "(         Origin: (" + str(px) + ", " + str(py) + ", " + str(pz) +
" ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Z"]
          Output "(         Axis: (" + str(px) + ", " + str(py) + ", " + str(pz) + "
) )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMin"]
          Output "(         Min level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMax"]
          Output "(         Max level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["ROut"]
          Output "(         Outer radius: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["RInn"]
          Output "(         Inner radius: " + str(pv) + " )"
        end
        8: begin !Prism
          Output "(         Type: Prism )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Z"]
          Output "(         Origin: (" + str(px) + ", " + str(py) + ", " + str(pz) +
" ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Z"]
          Output "(         Axis: (" + str(px) + ", " + str(py) + ", " + str(pz) + "
) )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMin"]
          Output "(         Min level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMax"]
          Output "(         Max level: " + str(pv) + " )"
        end
        9: begin !PolygonalPrism
          Output "(         Type: Polygonal prism )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Origin.Z"]
          Output "(         Origin: (" + str(px) + ", " + str(py) + ", " + str(pz) +
" ) )"
          px = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.X"]
          py = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Y"]
          pz = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Axis.Z"]
          Output "(         Axis: (" + str(px) + ", " + str(py) + ", " + str(pz) + "
) )"
          pv =
```

```
Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Int["CornerCount"]
          Output "(         Corner count: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["RInn"]
          Output "(         Inscribed radius: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["Angle"]
          Output "(         Angle around axis: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMin"]
          Output "(         Min level: " + str(pv) + " )"
          pv = Project.Ptr["Parts"].Item[i].Ptr["Workpiece"].Item[j].Flt["HMax"]
          Output "(         Max level: " + str(pv) + " )"
        end
        else begin
          Output "(         Type: Something else )"
        end
      end
    end
  end
subend
```

The result of this subroutine may be the following text block.

```
                    Results
( Part1.Workpiece )
(     Primitive 1 )
(         Type: Tube )
(         Origin: (0, 0, 117.5 ) )
(         Axis: (0, 0, 1 ) )
(         Min level: -149.5 )
(         Max level: 149.5 )
(         Outer radius: 66 )
(         Inner radius: 29 )
(     Primitive 2 )
(         Type: Cylinder )
(         Origin: (0, 0, 117.5 ) )
(         Axis: (0, 0, 1 ) )
(         Min level: -147.5 )
(         Max level: 147.5 )
(         Outer radius: 64 )
(     Primitive 3 )
(         Type: Turn envelope )
(         Origin: (0, 0, 117.5 ) )
(         Axis: (0, 0, 1 ) )
(     Primitive 4 )
(         Type: Box )
(         Min point: (-64, -64, -30 ) )
```

```
(         Max point: (64, 64, 265 ) )
(     Primitive 5 )
(         Type: Casting )
(         Stock: 1 )
(         Min point: (-65, -65, -31 ) )
(         Max point: (65, 65, 266 ) )
(     Primitive 6 )
(         Type: Faces )
(         Min point: (-54, -54, -30 ) )
(         Max point: (54, 54, 265 ) )
(     Primitive 7 )
(         Type: Polygonal prism )
(         Origin: (0, 0, 117.5 ) )
(         Axis: (0, 1, 0 ) )
(         Corner count: 6 )
(         Inscribed radius: 160.786 )
(         Angle around axis: 0 )
(         Min level: -64 )
(         Max level: 64 )
(     Primitive 8 )
(         Type: Empty workpiece )
```

Access from .NET

The project is available to a handler as the `CLDProject` object (`ICLDProject`); it is also passed to `OnStartProject` / `OnFinishProject`. `ICLDProject` implements the common named-property model, so the project properties are read exactly as with the sppx Project operator - `CLDProject.Ptr["Parts"]`, `CLDProject.Flt["..."]`, `CLDProject.Arr["Parts"][i]...` - with the same dotted/bracketed names. Besides the named properties it exposes:

.NET (**`ICLDProject`**)	Description
`CLDProject.ProjectName`	the project name (without a file path)
`CLDProject.FilePath`	the full path of the project file (empty if not yet saved)
`CLDProject.CLDFiles`	the list of CLData files (`ICLDFileList`, see CLData files)
`CLDProject.Operations`	the list of technological operations
`CLDProject.CLDSub`	the list of CLD-subroutines

.NET (`ICLDProject`)	Description
`CLDProject.Machine`	the machine info (name, type group, list of axes)
`CLDProject.FindCommand(type, startFile, startCommand)`	find a command of a type across the project

The same workpiece scan in .NET walks the same named tree (note the 0-based array indices - see the index-base note in Named CLData parameters). The complete method, ready to paste into a postprocessor - call it, for example, from `OnStartProject`:

```
// Outputs a comment for every element of every part's workpiece into the
NC-program.
public void PrintAllWorkpiece()
{
    var parts = CLDProject.Arr["Parts"];
    for (int i = 0; i <= parts.TopItem; i++)
    {
        nc.OutWithN("( Part" + (i + 1) + ".Workpiece )");
        var workpiece = parts[i].Arr["Workpiece"];
        for (int j = 0; j <= workpiece.TopItem; j++)
        {
            var wp = workpiece[j];
            nc.OutWithN("(     Primitive " + (j + 1) + " )");
            switch (wp.Int["PrimitiveType"])
            {
                case 0: // Unknown
                    nc.OutWithN("(         Type: Unknown )");
                    break;
                case 1: // Empty
                    nc.OutWithN("(         Type: Empty workpiece )");
                    break;
                case 2: // Faces
                    nc.OutWithN("(         Type: Faces )");
                    OutPoint(wp, "Min point", "Box.Min");
                    OutPoint(wp, "Max point", "Box.Max");
                    break;
                case 3: // Casting
                    nc.OutWithN("(         Type: Casting )");
                    nc.OutWithN("(         Stock: " + wp.Flt["Stock"] + " )");
                    OutPoint(wp, "Min point", "Box.Min");
                    OutPoint(wp, "Max point", "Box.Max");
                    break;
                case 4: // Box
                    nc.OutWithN("(         Type: Box )");
```

```
                    OutPoint(wp, "Min point", "Min");
                    OutPoint(wp, "Max point", "Max");
                    break;
                case 5: // RevBody
                    nc.OutWithN("(         Type: Turn envelope )");
                    OutPoint(wp, "Origin", "Origin");
                    OutPoint(wp, "Axis", "Axis");
                    break;
                case 6: // Cylinder
                    nc.OutWithN("(         Type: Cylinder )");
                    OutPoint(wp, "Origin", "Origin");
                    OutPoint(wp, "Axis", "Axis");
                    nc.OutWithN("(         Min level: " + wp.Flt["HMin"] + " )");
                    nc.OutWithN("(         Max level: " + wp.Flt["HMax"] + " )");
                    nc.OutWithN("(         Outer radius: " + wp.Flt["ROut"] + " )");
                    break;
                case 7: // Tube
                    nc.OutWithN("(         Type: Tube )");
                    OutPoint(wp, "Origin", "Origin");
                    OutPoint(wp, "Axis", "Axis");
                    nc.OutWithN("(         Min level: " + wp.Flt["HMin"] + " )");
                    nc.OutWithN("(         Max level: " + wp.Flt["HMax"] + " )");
                    nc.OutWithN("(         Outer radius: " + wp.Flt["ROut"] + " )");
                    nc.OutWithN("(         Inner radius: " + wp.Flt["RInn"] + " )");
                    break;
                case 8: // Prism
                    nc.OutWithN("(         Type: Prism )");
                    OutPoint(wp, "Origin", "Origin");
                    OutPoint(wp, "Axis", "Axis");
                    nc.OutWithN("(         Min level: " + wp.Flt["HMin"] + " )");
                    nc.OutWithN("(         Max level: " + wp.Flt["HMax"] + " )");
                    break;
                case 9: // PolygonalPrism
                    nc.OutWithN("(         Type: Polygonal prism )");
                    OutPoint(wp, "Origin", "Origin");
                    OutPoint(wp, "Axis", "Axis");
                    nc.OutWithN("(         Corner count: " + wp.Int["CornerCount"] +
" )");
                    nc.OutWithN("(         Inscribed radius: " + wp.Flt["RInn"] +
" )");
                    nc.OutWithN("(         Angle around axis: " + wp.Flt["Angle"] +
" )");
                    nc.OutWithN("(         Min level: " + wp.Flt["HMin"] + " )");
                    nc.OutWithN("(         Max level: " + wp.Flt["HMax"] + " )");
                    break;
                default:
```

```
                    nc.OutWithN("(         Type: Something else )");
                    break;
                }
            }
        }
    }

// Reads the .X / .Y / .Z children of a nested point property and prints them as
one comment.
private void OutPoint(INamedProperty item, string label, string basePath)
{
    double x = item.Flt[basePath + ".X"];
    double y = item.Flt[basePath + ".Y"];
    double z = item.Flt[basePath + ".Z"];
    nc.OutWithN("(         " + label + ": (" + x + ", " + y + ", " + z + " ) )");
}
```

See also

- Project information in CLData
- Named CLData parameters
- CLData files (CLDFile operator)
- CLData access functions and operators

GMA array

The GMA array is an sppx convenience for processing multi-axis MULTIGOTO movements: it presents the machine's controlled coordinates (with their new/previous values, output flags and angle-reduction results) in a single indexed structure.

> sppx only. There is no direct GMA counterpart in .NET. In a .NET postprocessor the multi-axis move is read from the strongly-typed MULTIGOTO command (`ICLDMultiGotoCommand`) - its `Axes` collection gives each axis and value - and the register handling that GMA performs in sppx is done through the .NET registers API. See the Access from .NET section of the MULTIGOTO command.

Access from sppx

Use the predefined GMA array to conveniently process multiaxis movement MULTIGOTO command in the technology program and in the mask of the command. GMA array is initialized directly before the MULTIGOTO technology command is processed.

Every item of GMA array corresponds to an controlled coordinate of the machine scheme. Items of the GMA array can be added manually using the Postprocessor generator interface or imported directly from CAM system machine scheme.

Optionally, coordinates can be added to GMA array during postprocessor execution using following operator syntax:

```
GMA(<AxisName> {,<RegisterName>})
```

Here AxisName is a legal expression that returns a string. The new coordinate name will be set to AxisName, use it to access the new item. Optional parameter RegisterName is the name of the register associated with the new coordinate.

Example

```
GMA("AxisXPos")
```

Use the name of a coordinate to access it's GMA properties:

```
GMA[<AxisName>].<Property>
```

Or for masks:

```
<Axisname>.<Property>
```

Here AxisName is a legal string expression that must be a valid coordinate name. In the following example the name of the coordinate AxisXPos is used to access the coordinate current value:

```
GMA["AxisXPos"].Vn ! Vn - coordinate current value
```

GMA Structure

GMA array item has the following structure:

```
GMA[<AxisName>].OutFlag
GMA[<AxisName>].Vn
GMA[<AxisName>].Vp
GMA[<AxisName>].Axis
GMA[<AxisName>].Reg
GMA[<AxisName>].TurnCount
GMA[<AxisName>].Dir
```

- `GMA[<AxisName>].OutFlag` is an Integer field that is assigned values 1 or 0. Use this flag to determine whether the coordinate named AxisName is listed in the given MULTIGOTO command. `GMA[<AxisName>].OutFlag` has the value of 1 (the flag is set) if there is a coordinate named AxisName in the given MULTIGOTO command, otherwise the flag is 0.
- `GMA[<AxisName>].Vn` is a Real field and it's value is the new value of the coordinate AxisName set by MULTIGOTO command. Use the Vn field to read or set the new coordinate value.
- `GMA[<AxisName>].Vp` is a Real field that contains the value of AxisName coordinate assigned by the previous MULTIGOTO command. Use the Vp field to read or set the previous coordinate value.
- `GMA[<AxisName>].Axis` is a String field that is the name of the coordinate and is equal to AxisName. The field is read-only.
- `GMA[<AxisName>].Reg` is a String field and it's value is the name of the register associated with the coordinate. The field is read-only.
- `GMA[<AxisName>].TurnCount` is an Integer field that is assigned the number of full turns of a rotatable axis. The field is read-only.
- `GMA[<AxisName>].Dir` is an Integer value that is assigned +1 if the value of the coordinate is increasing, otherwise it is set to -1. The field is read-only.

Use TurnCount and Dir fields to process the movement of rotatable axes. CAM system passes the movements of rotatable axes in absolute values, whereas some NC-systems accept only values in bound ranges like 0° to 360° or 180° to -180° for those coordinates. Use the Machine axes parameters dialog to define the bounds and ranges parameters of rotatable axes. If angle reduction is defined for a given

coordinate then when processing MULTIGOTO command the system will reduce the coordinate value Vn to specified range, assign the TurnCount field the number of full revolutions of the coordinate and set the sign of the Dir field depending on the previous (not reduced) value of the coordinate.

Here are a few examples illustrating how the system initializes the properties of GMA array items when processing MULTIGOTO technology command. Suppose rotatable axis C (AxisCPos) is bound to values from 0° to 360°.

1. Command:

```
MULTIGOTO COUNT 4, AxisXPos(18) 80, AxisYPos(20) 19.877,
AxisZPos(22) -200, AxisCPos(24) 0
```

GMA array for the MULTIGOTO command:

Coordinate name	OutFlag	Vn	Vp	Axis	Reg	TurnCount	Dir
AxisXPos	1	80	-//-//-	"AxisXPos"	"X"	-//-//-	-//-//-
AxisYPos	1	19.877	-//-//-	"AxisYPos"	"Y"	-//-//-	-//-//-
AxisZPos	1	-200	-//-//-	"AxisZPos"	"Z"	-//-//-	-//-//-
AxisCPos	1	0	-//-//-	"AxisCPos"	"C"	-//-//-	-//-//-

"-//-//-" sign indicates that the values of the properties of coordinates cannot be determined because there is no information of the previous command.

1. Command:

```
MULTIGOTO COUNT 1, AxisCPos(24) 270
```

GMA array for the MULTIGOTO command:

Coordinate name	OutFlag	Vn	Vp	Axis	Reg	TurnCount	Dir
AxisXPos	0	80	80	"AxisXPos"	"X"	-//-//-	-//-//-
AxisYPos	0	19.877	19.877	"AxisYPos"	"Y"	-//-//-	-//-//-
AxisZPos	0	-200	-200	"AxisZPos"	"Z"	-//-//-	-//-//-
AxisCPos	1	270	0	"AxisCPos"	"C"	0	+1

1. Command:

```
MULTIGOTO COUNT 2, AxisZPos(22) 250, AxisCPos(24) -745
```

GMA array for the MULTIGOTO command:

Coordinate name	OutFlag	Vn	Vp	Axis	Reg	TurnCount	Dir
AxisXPos	0	80	80	"AxisXPos"	"X"	-//-//-	-//-//-

Coordinate name	OutFlag	Vn	Vp	Axis	Reg	TurnCount	Dir
AxisYPos	0	19.877	19.877	"AxisYPos"	"Y"	-//-//-	-//-//-
AxisZPos	1	250	-200	"AxisZPos"	"Z"	-//-//-	-//-//-
AxisCPos	1	335	270	"AxisCPos"	"C"	2	-1

In the last example the system applied angle reduction for the `AxisCPos(C)` coordinate. The value passed by CAM system "-745 °" is reduced to the range from 0° to 360° by the following rules:

```
TurnCount = |(Vn' - Vp') \ 360|
TurnCount = |(-745 -270) \ 360| = 2
Dir = Sgn(Vn'-Vp')
Dir = Sgn(-745 -270) = -1
Vn = Vn' - (Vn' \ 360)*360, if Vn' > 360
Vn = Vn' - (Vn' \ 360)*360 + 360, if Vn' < 0
Vn = -745 - (-745\360)*360 + 360 = 335
```

`Vn'` - the current not reduced value of the coordinate, `Vp'` - previous not reduced value of the coordinate.

The following is an example of a simple MULTIGOTO command program that uses the GMA array:

```
program MultiGoto
  ! Use the GMA fields to assign
  ! respective registers their new values
  if GMA["AxisXPos"].OutFlag>0 then
    X = GMA["AxisXPos"].Vn
  if GMA["AxisYPos"].OutFlag>0 then
    Y = GMA["AxisYPos"].Vn
  if GMA["AxisZPos"].OutFlag>0 then
    Z = GMA["AxisZPos"].Vn
  if GMA["AxisCPos"].OutFlag>0 then
    C = GMA["AxisCPos"].Vn
  ! Output the formed values into the NC-program
  OutBlock
end
```

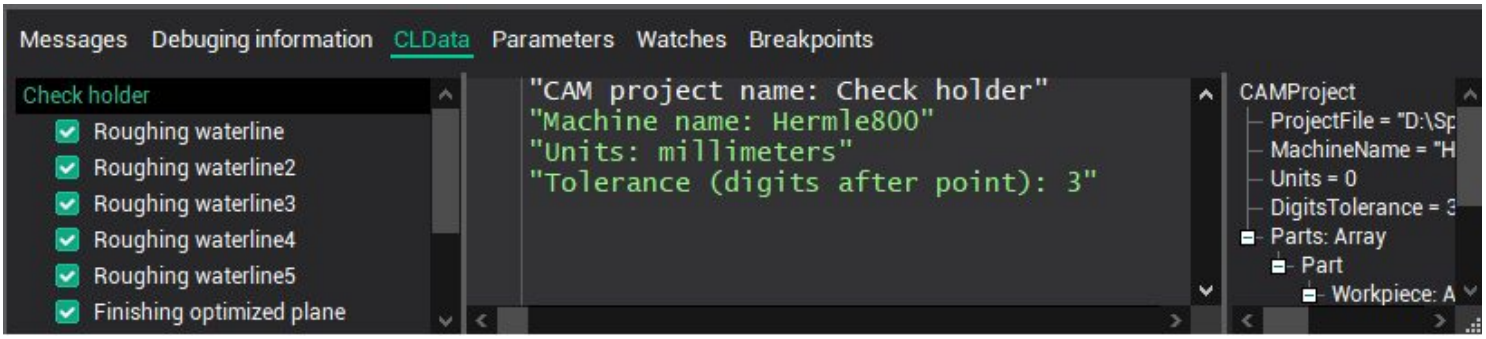
See also

- Predefined functions
- MULTIGOTO command
- Masks
- CLData access functions and operators

Project information in CLData

The information about the CAM system project placed inside CLData available through the Project operator.

You can display it on the screen if on the CLData tab select the first item with the name of the project. Properties will be shown in the parameters tree on the right.



The list of possible properties described in tables below. Information about the properties structure, presented as an XML file, is available in the CAM system installation folder in a subfolder Supplement\Operations\CLData.xml.

Type	Description	Subfield description
Equipment: ComplexType	"Equipment" - this is a keyword that allows you to access machine properties that have been loaded from a CAM system machine schema file. The list of properties inside this node can vary depend on the machine schema file.	
	Schemas: Array	Equipment.Ptr["Schemas"] - The list of machine schemas used in the project (usually only one schema).
MachineSchema: ComplexType	Equipment.Ptr["Schemas(1)"] - root element of the machine schema property nodes, Usually the name of this property uniquely identifies the machine schema ID:	

Type	Description	Subfield description
	Equipment.Ptr["Schemas(1)"].Name - unique identifier of the machine schema.	
Name: String	Equipment.Str["Schemas(1).Name"] - user friendly name of the machine.	
ProjectFile: String	Project.Str["ProjectFile"] - the full path to the CAM system project file name. Can be empty for new projects when they are not saved.	
MachineName: String	Project.Str["MachineName"] - the name of the machine from the machine schema file.	
Units: Integer	Project.Int["Units"] - The project's units system: 0 - metric (mm), 1 - imperial (inch).	
DigitsTolerance: Integer	Project.Int["DigitsTolerance"] - the number of digits for the project's tolerance. Typical value is 3 (which means 0.001). It can be changed in CAM system's settings window.	
SetupStages: Array	Project.Ptr["SetupStages"] - The list of Setup stage groups.	
	SetupStage: ComplexType	Project.Ptr["SetupStages"].Item[index] - Setup stage group information.
Index: Integer	Project.Ptr["SetupStages"].Item[index].Int["Index"] - The index of the setup stage group in the list of setup stages in CAM system project.	

Type	Description	Subfield description
Name: String	Project.Ptr["SetupStages"].Item[index].Str["Name"] - The name of the setup stage group in CAM system project.	
Parts: Array	Project.Ptr["Parts"] - The list of structures that contain information about the part. The count of items depend on count of parts in the project.	
Part: ComplexType	Project.Ptr["Parts"].Item[index] - the structure that contain information about one part.	
Name: String	Project.Ptr["Parts"].Item[index].Str["Name"] - The Part name (group name) that is set in CAM system.	
SetupStageIndex: Integer	Project.Ptr["Parts"].Item[index].Int["SetupStageIndex"] - The index of the setup stage group in the list of setup stages in CAM system project inside which the Part is.	
PartIndex: Integer	Project.Ptr["Parts"].Item[index].Int["PartIndex"] - Index of the Part in the list of CAM system's project parts list	
PartNumber: Integer	Project.Ptr["Parts"].Item[index].Int["PartNumber"] - Part number that is explicitly set by the user in the part parameters of CAM system	
WorkpieceConnector: String	Project.Ptr["Parts"].Item[index].Str["WorkpieceConnector"] - The	

Type	Description	Subfield description
	textual identifier of the workpiece connector in which the Part is fixed. This identifier depend on the Workpiece connector node id inside machine schema file.	
WCSList: Array	Project.Ptr["Parts"].Item[index].Ptr["WCSList"] - The list of workpiece coordinate systems that are used when machining this part (for example G54, G55, etc.).	
WCS: ComplexType	A structure for describing the Workpiece coordinate system	
Number: Integer	The workpiece coordinate system number	
Location: ComplexType	Geometrical parameters of the WCS	
vX: Complex type	X axis vector of the WCS in the base coordinate system	
X: Double	X coordinate of the vector	
Y: Double	Y coordinate of the vector	
Z: Double	Z coordinate of the vector	
vY: Complex type	Y axis vector of the WCS in the base coordinate system	
X: Double	X coordinate of the vector	
Y: Double	Y coordinate of the vector	
Z: Double	Z coordinate of the vector	

Type	Description	Subfield description
vZ: Complex type	Z axis vector of the WCS in the base coordinate system	
X: Double	X coordinate of the vector	
Y: Double	Y coordinate of the vector	
Z: Double	Z coordinate of the vector	
vT: Complex type	Original point of the WCS in the base coordinate system	
X: Double	X coordinate of the point	
Y: Double	Y coordinate of the point	
Z: Double	Z coordinate of the point	
PartBox: ComplexType	Project.Ptr["Parts(1).PartBox"] - A structure for describing the overall dimensions of the part.	
Empty: Boolean	The state of the part dimensions: false (0) - dimensions are set and have correct values, true (1) - dimension are empty (for example when part geometry absent).	
Min: ComplexType	Left bottom corner point of the overall box of the Part.	
X: Double	X coordinate of the point	
Y: Double	Y coordinate of the point	
Z: Double	Z coordinate of the point	
Max: ComplexType	Right top corner point of the overall box of the Part.	

Type	Description	Subfield description
X: Double	X coordinate of the point	
Y: Double	Y coordinate of the point	
Z: Double	Z coordinate of the point	
WorkpieceBox: ComplexType	Project.Ptr["Parts(1).WorkpieceBox"] - A structure for describing the overall dimensions of the workpiece.	
Empty: Boolean	The state of the workpiece dimensions: false (0) - dimensions are set and have correct values, true (1) - dimension are empty (for example when workpiece geometry absent).	
Min: ComplexType	Left bottom corner point of the overall box of the Workpiece.	
X: Double	X coordinate of the point	
Y: Double	Y coordinate of the point	
Z: Double	Z coordinate of the point	
Max: ComplexType	Right top corner point of the overall box of the Workpiece.	
X: Double	X coordinate of the point	
Y: Double	Y coordinate of the point	
Z: Double	Z coordinate of the point	
Workpiece: Array	Project.Ptr["Parts"].Item[index].Ptr["Workpiece"] - the list of structures that contain information about workpiece	

Type	Description	Subfield description
	items that correspond to the part with defined index.	
Item: TCLDModelItem	Project.Ptr["Parts"].Item[index].Ptr["Workpiece"].Item[index2] - the structure that contain information about the one workpiece item. The composition of the properties depends on the type of element. All of them are inheritors from complex type TCLDModelItem, so all elements have one common property - the primitive type. See Possible workpiece items table below.	
PrimitiveType: Integer	Project.Ptr["Parts"].Item[index].Ptr["Workpiece"].Item[index2].Int["PrimitiveType"] - the type of the geometrical primitive: 0 - Unknown, 1 - Empty workpiece 2 - Faces (reference to the CAD model) 3 - Casting 4 - Box 5 - Turn envelope 6 - Cylinder 7 - Tube 8 - Prism 9 - Polygonal prism	

Possible workpiece items

Type	Description			
EmptyWorkpiece: TCLDModelItem	Element of the workpiece, which says that the processing begins with a void. For example, if we grow a part from scratch by			

Type	Description			
	methods of additive processing			
Box: TCLDModelItem	The workpiece item representing the parallelepiped given by two points.			
Min: ComplexType	Left bottom corner point of the box.			
X: Double	X coordinate of the point			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
Max: ComplexType	Right top corner point of the box			
X: Double	X coordinate of the point			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
RevBody: TCLDModelItem	The workpiece item representing the turn envelope defined by the axis around which the rotation is performed and a set of geometric			

Type	Description			
	primitives that rotate.			
Origin: ComplexType	Start point of the rotation axis			
X: Double	X coordinate of the point			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
Axis: ComplexType	Vector that represents the axis of rotation			
X: Double	X coordinate of the vector			
Y: Double	Y coordinate of the vector			
Z: Double	Z coordinate of the vector			
Cylinder: RevBody	The workpiece item representing cylinder along defined axis. It is a successor of the RevBody type, therefore it contains all its properties, which are described above.			
HMin: Double	Start level of the cylinder. This is			

Type	Description			
	the distance along the axis, starting from the point of origin.			
HMax: Double	Finish level of the cylinder. This is the distance along the axis, starting from the point of origin.			
ROut: Double	Outer radius of the cylinder.			
Tube: Cylinder	The workpiece item representing tube along defined axis. It is a successor of the Cylinder type, therefore it contains all its properties, which are described above.			
RInn: Double	Inner radius (hole radius) of the tube.			
Prism: TCLDModelItem	The workpiece item representing the prism (extrusion item) defined by the axis along which the extrusion is performed and profile to extrude.			

Type	Description			
Origin: ComplexType	Start point of the extrusion axis			
X: Double	X coordinate of the point			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
Axis: ComplexType	Vector that defines the direction of extrusion			
X: Double	X coordinate of the vector			
Y: Double	Y coordinate of the vector			
Z: Double	Z coordinate of the vector			
HMin: Double	Start level of the prism. This is the distance along the axis, starting from the point of origin.			
HMax: Double	Finish level of the prism. This is the distance along the axis, starting from the point of origin.			
PolygonalPrism: Prism	The workpiece item representing regular polygonal			

Type	Description			
	prism along defined axis. It is a successor of the Prism type, therefore it contains all its properties, which are described above.			
RInn: Double	The radius of the inscribed circle of the polygon			
Angle: Double	The angle of rotation of the polygon around the axis			
CornerCount: Integer	The corners count of the polygon			
Faces: TCLDModelItem	The workpiece item representing the general form body specified by the CAD model reference.			
Box: ComplexType	The box around CAD-model. Lets you know the overall dimensions.			
Min: ComplexType	Left bottom corner point of the box.			
X: Double	X coordinate of the point			

Type	Description			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
Max: ComplexType	Right top corner point of the box			
X: Double	X coordinate of the point			
Y: Double	Y coordinate of the point			
Z: Double	Z coordinate of the point			
Casting: TCLDModelItem	The workpiece item representing the body, which is obtained by adding an additional stock to the body of the general form specified by the CAD model reference.			
	Box: ComplexType	The box around CAD-model with stock. Lets you know the overall dimensions.		
		Min: ComplexType	Left bottom corner point of the box.	
			X: Double	X coordinate of the point

Type	Description			
			Y: Double	Y coordinate of the point
			Z: Double	Z coordinate of the point
		Max: ComplexType	Right top corner point of the box	
			X: Double	X coordinate of the point
			Y: Double	Y coordinate of the point
			Z: Double	Z coordinate of the point
	Stock: Double	The value of stock.		

See also

- CLData

Technology commands description

The list of CLData technology commands:

Command	Code	Description
AXESBRAKE	2012	Machine axes brakes control
CIRCLE	15000	Circle movement
CLAMP	1074	Fixtures clamp/unclamp
COMMENT	1065	Commentaries
COOLNT	1030	Cooling
CUTCOM	1007	Tool compensation
CYCLE	1054	Hole machining canned cycles
DELAY	1010	Delay
EDMMOVE	16001	EDM movement
EFFECTOR	2019	End effector activation (EFFECTOR)
EXTCYCLE	1053	Extended cycle
FEDRAT	1009	Feedrate
FINI	14000	Ending record
FROM	5003	Original point
GOHOME	17	Return to original position
GOTO.abs	5005	Linear transition
HEAD	1002	Spindle head number
INSERT	1046	Insertion
INTERPOLATION	1047	Interpolation mode
LOADTL	1055	Tool loading

Command	Code	Description
MULTIARC	9014	Multiaxis circle movement
MULTIGOTO	9010	Multi coordinate movement
OPSKIP	1012	Optional skipping
OPSTOP	2003	Auxiliary stop
ORIGIN	1027	Original coordinates
PALETA	1001	Palette changing
PARTNO	1045	Part number
PhysicGOTO	9013	Physical machine axes movement
PLANE	99	Working plane
POWER	2018	Power adjustment (POWER)
PPFUN	1079	Postprocessor function
PPRINT	1044	Postprocessor printing
RAPID	2005	Rapid feedrate
ROTABL	1026	Table rotation
SAFPOS	1094	Tool change point
SELWORKPIECE	2011	Active workpiece holder selection
SINGLETHREAD	1037	Singlethread command
SPINDL	1031	Spindle
STOP	2002	Stop
STRUCTURE	2015	Structure command
SYNCAXES	2016	Axes movement syncronization

Command	Code	Description
TAKEOVER	2010	Workpiece takeover
TLCONTACT	2017	Tool contact command
WAIT	2014	Waiting for synchronization point

Additional words:

N	Word	Code	Description
1	BOTH	83	Both
2	BRKCHP	288	Chip breaking
3	CCLW	59	Counter clockwise
4	CLW	60	Clockwise
5	CUTS	511	Cuts
6	DEEP	153	Deep drilling
7	DEPTH	510	Depth
8	DRILL	163	Drill
9	DWELL	279	Dwell
10	FACE	81	G82 canned cycle
11	FINCUT	512	Final cut
12	INCR	66	Increment
13	LENGTH	9	Length
14	MMPM	315	Millimeters per minute
15	MMPR	316	Millimeters per revolution
16	MULTRD	119	Multi-start thread
17	OFF	72	Turn off

N	Word	Code	Description
18	ON	71	Turn on
19	ORIENT	246	Orientation
20	TAP	168	Tapping
21	XYPLAN	33	XY plane
22	YZPLAN	37	YZ plane
23	ZXPLAN	41	ZX plane
24	R	23	Radius
25	RGT	24	Right tool positioning
26	LEFT	8	Left tool positioning

Global handlers (not CLData commands)

Handlers invoked outside the per-command dispatch - listed here so all postprocessor handlers can be found in one place:

Handler	sppx	.NET	Description
Output filter	`sub Filter`	`OnFilterString`	Runs on every NC-program line, whatever produced it.

See also

- CLData

AXESBRAKE - Machine axes brakes control

Command caption

How the command appears in the CLData command list:

```
AXESBRAKE COUNT N, Axis1Pos(n1) State1, Axis2Pos(n2) State2, ...,
AxisNPos(nN) StateN
```

Use AXESBRAKE command to turn on and off axes brakes if there are axes brakes on the machine. Most commonly this command is used to turn on/off spindle brake (Axis C) on the lathe-milling machines when switching from lathe to milling and vice versa. When performing lathe machining the spindle rotates at high speed, and when milling the spindle is fixed by the brake at specific angle. Using brakes helps to increase the spindle load and reduces the spindle precision positioning mechanisms wear. Also brakes are used on milling machines rotating axes to perform indexed machining.

Access from sppx

Handler: `program AxesBrake`

Parameter	CLD array	Description
N	CLD[1]	Number of axes affected by this command.
n1	CLD[2]	Index of axis named Axis1Pos in the coordinates list.
State1	CLD[3]	New state of Axis1Pos axis: ON(71) - brake is on, OFF(72) - brake is off.
n2	CLD[4]	Index of Axis2Pos axis in the coordinates list.
State2	CLD[5]	New state of Axis2Pos axis: ON(71) - brake is on, OFF(72) - brake is off.
...	...	...
nN	CLD[2*N]	Index of AxisNPos axis in the coordinates list.

Parameter	CLD array	Description
StateN	CLD[2*N+1]	New state of AxisNPos axis: ON(71) - brake is on, OFF(72) - brake is off.

Parameters available through the cmd operator

TCLDAxesBrake: ComplexType	The command to switch the states of axes brake.
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - An array of structures such as AxisBrake. Thus, one command can change the brake state for several axes.
AxisBrake: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - Separate element of the Axes array. Contains the state of one machine axis brake. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - The machine axis identifier, for which is given a new brake state. Determined by the machine schema.
BrakeState: Integer	cmd.Int["Axes(AxisName).BrakeState"] - the new state of the machine axis brake: 0 - Off, 1 - On.

Command syntax allows changing states of multiple axes brakes at once. List of coordinates, which names appear in this command are defined by the CAM system machine scheme.

Here are two simple examples of programs handlers for this command.

```
program AxesBrake
  Index: Integer ! Loop counter
  AxisNumber: Integer ! Index of an axis in the list of machine axes
  BrakeState: Integer ! New state of the axis brake
  Index = 1
  while Index<=CLD[1] do begin
    AxisNumber = CLD[2*Index]
    BrakeState = CLD[2*Index+1]
    case AxisNumber of
```

```
      4: begin ! AxisAPos(A) index in the list of machine axes
        if BrakeState=71 then Output "M680" ! Turn axis A brake on
        else Output "M690" ! Turn axis A brake off
      end
      6: begin ! AxisAPos(A) index in the list of machine axes
        if BrakeState=71 then Output "M68" ! Turn axis C brake on
        else Output "M69" ! Turn axis C brake off
      end
    end
    Index = Index + 1
  end
end
```

Another example with cmd operator using.

```
program AxesBrake
  if cmd.Ptr["Axes(AxisAPos)"]<>0 then begin ! Axis A is present in this command
    if cmd.Int["Axes(AxisAPos).BrakeState"]=1 then Output "M680" ! Turn axis A
brake on
    else Output "M690" ! Turn axis A brake off
  end
  if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin ! Axis C is present in this command
    if cmd.Int["Axes(AxisCPos).BrakeState"]=1 then Output "M68" ! Turn axis C
brake on
    else Output "M69" ! Turn axis C brake off
  end
end
```

Access from .NET

Handler:

```
public override void OnAxesBrake(ICLDAxesBrakeCommand cmd, CLDArray cld)
```

Inheritance: `ICLDAxesBrakeCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.AxesCount`	`int`	The count of axes inside the "Axis" list.
`cmd.Axis`	`ICLDBrakeAxesIndexer`	The list of axes brake states Axis[i] or Axis["AxisCPos"].
`cmd.Axes`	`IEnumerable`	Axes' enumerator object for the brakes to be possible to use covenient foreach syntax.
`cmd.HasAxis(...)`	`bool`	Returns "True" if the "Axis" list contains the axis with the specified textual ID. Otherwise returns "False".

Example - `OnAxesBrake` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnAxesBrake(ICLDAxesBrakeCommand cmd, CLDArray cld)
{
    foreach(CLDAxisBrake ax in cmd.Axes) {
        if (ax.IsA) {
            if (ax.StateIsOn)
                nc.MABrake.v = 593;
            else
                nc.MABrake.v = 592;
        } else if (ax.IsB) {
            if (ax.StateIsOn)
                nc.MBBrake.v = 595;
            else
                nc.MBBrake.v = 594;
        } else if (ax.IsC) {
            if (ax.StateIsOn)
                nc.MCBrake.v = 597;
            else
                nc.MCBrake.v = 596;
        }
    }
    if (nc.MABrake.Changed || nc.MBBrake.Changed || nc.MCBrake.Changed)
        nc.Block.Out();
}
```

See also

- CLData access model

CIRCLE - Circle movement

Command caption

How the command appears in the CLData command list:

```
CIRCLE XC xc, YC yc, ZC zc, R r, XE xe, YE ye, ZE ze
```

CIRCLE command represents movement along an arc. CIRCLE parameters are arc center, arc end point and arc radius. The sign of the radius defines the arc's direction. If the radius is positive then the arc has counter clockwise direction (G03 circle interpolation). If the radius is negative then the arc has clockwise direction (G02 circle interpolation). The arc's plane (G17/G18/G19) is defined by the current machining plane, set by the PLANE command.

Access from sppx

Handler: `program Circle`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
xc yc zc	CLD[1] CLD[2] CLD[3]	CLD.Xc CLD.Yc CLD.Zc	X, Y and Z coordinates of the arc center.
r	CLD[4]	CLD.R	Arc radius. If CLD.R > 0 then the arc is counter clockwise (G03), else if CLD.R < 0 then the arc is clockwise (G02).
xe ye ze	CLD[5] CLD[6] CLD[7]	CLD.Xe CLD.Ye CLD.Ze	X, Y and Z coordinates of the arc end point.
xs ys zs	CLD[8] CLD[9] CLD[10]	CLD.Xs CLD.Ys CLD.Zs	X, Y and Z coordinates of the arc start point
XcInc YcInc	CLD[11] CLD[12]	CLD.XcInc CLD.YcInc	X, Y and Z coordinates of the arc center point

Parameter	CLD array	CLD name	Description
ZcInc	CLD[13]	CLD.ZcInc	relative to the start point (I, J and K)
RAbs	CLD[14]	CLD.RAbs	Absolute value of the arc radius
RIso	CLD[15]	CLD.RIso	Arc radius in ISO variant - positive if angle less than 180 degrees and negative in other case
Dir	CLD[16]	CLD.Dir	Direction of the arc: 2-clockwise, 3-counterclockwise relative to the ISO plane G17-G19
Plane	CLD[17]	CLD.Plane	Plane of the arc, can take 6 values: 17, 18, 19, -17, -18, -19. Corresponds to the ISO planes G17-G19. A negative value means that the normal of the arc plane is inverse relative to the standard ISO plane.
Length	CLD[18]	CLD.Length	Size of the arc in current linear units
Ang	CLD[19]	CLD.Ang	Angle of the arc in degrees (0 to 360 degrees)
SAng	CLD[20]	CLD.SAng	Start angle of the arc relative to the first axis of the current plane

Parameter	CLD array	CLD name	Description
EAng	CLD[21]	CLD.EAng	End angle of the arc relative to the first axis of the current plane
HelixAng	CLD[22]	CLD.HelixAng	Angle of the helix in degrees. zero for planar circle, positive value for rising helix, negative value for descending helix.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Circle
  RSign: Integer

  if cld[4] > 0 then INTERP_ = 3 else INTERP_ = 2   ! G3/G2
  X = cld[5]                                        ! X,Y,Z in absolutes
  Y = cld[6]
  Z = cld[7]
  if CirclesThroughRadius>0 then begin
    call CheckCircleAngle(RSign)
    Rc = RSign*abs(CLD.R); Rc@ = MaxReal
  end else begin
    XC_ = cld[1] - XT_                              ! I,J,K in increments always
    YC_ = cld[2] - YT_
    ZC_ = cld[3] - ZT_
    XC_@ = MaxReal; ZC_@ = MaxReal
    if OperationIsLathe<>1 then
      YC_@ = MaxReal
    case Plane of
      17: ZC_@ = ZC_;
      18: YC_@ = YC_;
      19: XC_@ = XC_;
    end
  end
  case Plane of
    17: begin
      X@ = MaxReal;
      Y@ = MaxReal
    end
    18: begin
      X@ = MaxReal;
```

```
      Z@ = MaxReal
    end
    19: begin
      Y@ = MaxReal
      Z@ = MaxReal
    end
  end
  if IsRotaryInterpolation = 2 then begin
    Y = -Y/CylR/Pi*180
  end
  OutBlock                    ! output to NC block
  XT_ = cld[5]                ! save current coordinates
  YT_ = cld[6]
  ZT_ = cld[7]
end
```

Access from .NET

Handlers (the specific handler plus the movement-group brackets that fire for every motion command):

```
public override void OnCircle(ICLDCircleCommand cmd, CLDArray cld)
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
before any movement
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)    //
after any movement
```

Inheritance: `ICLDCircleCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement. *(inherited from `ICLDMotionCommand`)*
`cmd.SP`	`TInp3DPoint`	Arc start point (X, Y, Z).
`cmd.Center`	`TInp3DPoint`	Arc center point (X, Y, Z), absolute.

Property	Type	Description
`cmd.IncCenter`	`TInp3DPoint`	Arc center point (I, J, K), incremental to the start point.
`cmd.R`	double	Arc radius; positive for CCW (G03), negative for CW (G02).
`cmd.RAbs`	double	Absolute value of the arc radius.
`cmd.RIso`	double	ISO-standard radius: positive if the arc angle is < 180 deg, negative if > 180 deg.
`cmd.Dir`	int	ISO direction code: `2` = CW (G2), `3` = CCW (G3).
`cmd.Plane`	int	Plane of the arc: `17, 18, 19, -17, -18, -19` (ISO G17-G19; negative = inverted normal).
`cmd.Length`	double	Length of the arc in current linear units.
`cmd.Ang`	double	Angle of the arc in degrees (0 to 360).
`cmd.SAng` `cmd.EAng`	double	Start / end angle of the arc relative to the first axis of the current plane.
`cmd.HelixAng`	double	Helix angle in degrees (0 planar, >0 rising, <0 descending).

Example (from the Fanuc 30i mill postprocessor):

```
public override void OnCircle(ICLDCircleCommand cmd, CLDArray cld)
{
    nc.GInterp.Show(cmd.Dir);                                   // G02 / G03
    nc.X.v = cmd.EP.X;  nc.Y.v = cmd.EP.Y;  nc.Z.v = cmd.EP.Z;
    switch (Abs(cmd.Plane))                                     // arc center as
incremental I, J, K
    {
        case 17:
```

```
            nc.I.v = cmd.IncCenter.X;  nc.J.v = cmd.IncCenter.Y;
            nc.Block.Show(nc.X, nc.Y, nc.I, nc.J);
            break;
        case 18:
            nc.I.v = cmd.IncCenter.X;  nc.K.v = cmd.IncCenter.Z;
            nc.Block.Show(nc.X, nc.Z, nc.I, nc.K);
            break;
        case 19:
            nc.J.v = cmd.IncCenter.Y;  nc.K.v = cmd.IncCenter.Z;
            nc.Block.Show(nc.Y, nc.Z, nc.J, nc.K);
            break;
    }
}
```

See also

- CLData access model

CLAMP - Fixtures clamp

Use CLAMP command to control part fixture devices (such as chucks, vices, collets, etc.) in the pick-and-place, takeover operations.

Command: CLAMP ID(N), ON(71)|OFF(72) DIR(N)

Access from sppx

Handler: `program Clamp`

Parameter	CLD array	CLD name	Description
ID	CLD[1]	CLD.ID	Unique identifier of the clamping device you need to switch. Depend on the equipment settings inside a CAM project.
IsOn	CLD[2]	CLD.IsOn	New state of the clamping device: 71 - clamp is on, the workpiece is fixed in the clamping device. 72 - clamp is off. the workpiece is released.
Dir	CLD[3]	CLD.Dir	Clamping direction (-1; 0; +1) - direction of the clamping jaw movement (sign of jaw axis coordinate change).

Here is a simple example of programs handlers for this command.

```
program Clamp
  if CLD[2] = 72 then begin
    call OutPause(1)
    if CLD[1] = 2 then begin
      Mchuck = 111; Mchuck@ = Maxreal
      Formblock
      Outstr$ = Outstr$ + " (OTKRYT-PROTIVOSHPINDEL)"
```

```
      Output Outstr$
      Formblock
      Outstr$ = Outstr$ + " M114 (AIR ON)"
      Output Outstr$
      call OutPause(1)
      Formblock
      Outstr$ = Outstr$ + " M115 (AIR OFF)"
      Output Outstr$
    end else begin
      Mchuck = 11; Mchuck@ = Maxreal
      Formblock
      Outstr$ = Outstr$ + " (OTKRYT-GLAVNYi SHPINDEL)"
      Output Outstr$
      Formblock
      Outstr$ = Outstr$ + " M14 (AIR ON)"
      Output Outstr$
      call OutPause(1)
      Formblock
      Outstr$ = Outstr$ + " M15 (AIR OFF)"
      Output Outstr$
    end
  end else begin
    if CLD[1] = 2 then begin
      Mchuck = 110; Mchuck@ = Maxreal
      Formblock
      Outstr$ = Outstr$ + "(ZAKRIT-PROTIVOSHPINDEL)"
    end else begin
      Mchuck = 10; Mchuck@ = Maxreal
      Formblock
      Outstr$ = Outstr$ + "(ZAKRIT-GLAVNIY SHPINDEL)"
    end
    Output Outstr$
    call OutPause(1)
  end
end
```

Access from .NET

Handler:

```
public override void OnClamp(ICLDClampCommand cmd, CLDArray cld)
```

Inheritance: `ICLDClampCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Example - `OnClamp` (from the Goodway SW postprocessor):

```
public override void OnClamp(ICLDClampCommand cmd, CLDArray cld)
{
    base.OnClamp(cmd, cld);
    bool MainSp = cmd.ClampID==1;
    outClamp(nc, MainSp, cmd.IsOn, 0.2);
}
```

See also

- CLData access model

COMMENT - Commentaries

Command caption

How the command appears in the CLData command list:

```
COMMENT "........"
```

COMMENT outputs comments line into the NC-program. The comment is put into CLDATA$ variable. Comments can be transliterated.

Access from sppx

Handler: `program Comment`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Comment
  if CLDATA$<>"" then begin
    if pos("@", CLDATA$) = 1  then begin
      CLDATA$ = ""!Copy(CLDATA$, 2, Len(CLDATA$))
    end
    if CLDATA$<>"" then
      Output "( "+CLDATA$+" )"
  end
end
```

Access from .NET

Handler:

```
public override void OnComment(ICLDCommentCommand cmd, CLDArray cld)
```

Inheritance: `ICLDCommentCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.Text`	`string`	Textual string of a comment.
`cmd.UpperCaseText`	`string`	Textual string of a comment in upper case.
`cmd.IsOperationName`	`bool`	Returns "true" if this command is a first COMMENT command in a file of operation, that contains the name of this operation. If it's another type of comment, this flag will contain "false".
`cmd.IsToolName`	`bool`	Returns "true" if this command is a special COMMENT command that contains the name of the tool. If it's another type of comment, this flag will contain "false".

Example - `OnComment` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnComment(ICLDCommentCommand cmd, CLDArray cld)
{
    if (!(cmd.IsOperationName || cmd.IsToolName)) {
        nc.OutWithN("( " + Transliterate(cmd.CLDataS) + " )");
    }
}
```

See also

- CLData access model

COOLNT - Cooling

Command caption

How the command appears in the CLData command list:

```
COOLNT ON(71)|OFF(72), N n
```

COOLNT command turns on or off the machine cooling systems. The first parameter specifies command type: activation or deactivation. The second parameter is the tube number which is affected by the COOLNT command.

Access from sppx

Handler: `program Coolnt`

Parameter	CLD array	CLD name	Description
ON(71) OFF(72)	CLD[1]	CLD.OnOff	Command type: ON(71) – activate coolant, OFF(72) – deactivate coolant.
n	CLD[2]	CLD.N	Coolant tube number: 1 – flood, 2 – mist, 3 – tool.

To create ISO postprocessor using masks use ISO.M values which correspond to parameters:

CLD parameter	ISO value
CLD[1] = 71	ISO.M = 8
CLD[1] = 72	ISO.M = 9

Example (from the Fanuc (30i)_Mill postprocessor):

```
program CoolNt
  if cld[1] = 71 then begin
    if cld[2] = 1 then begin
      Mc = 8
    end else if cld[2] = 2 then begin
```

```
      Mc = 8
    end else if cld[2] = 3 then begin
      Mc = 8
    end else begin
      Mc = 8
    end
  end else begin
    Mc = 9     ! On/Off coolant
    OutBlock
  end
end
```

Access from .NET

Handler:

```
public override void OnCoolant(ICLDCoolantCommand cmd, CLDArray cld)
```

Inheritance: `ICLDCoolantCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Returns "True" if it's a switch on command and you need to switch ON all tubes marked in the TubeIsOn[] list.
`cmd.IsOff`	`bool`	Returns "True" if it's a switch off command and you need to switch OFF all tubes marked in the TubeIsOn[] list.
`cmd.TubeCount`	`int`	Returns the count of tubes presented in the TubeIsOn[] list.
`cmd.TubeIsOn`	`ICLDCoolantTubeIndexer`	Allows you to determine whether or not to switch the state for the tube with a given index. The

Member	Type	Description
		index must be between 0..31 (TubeCount-1).
`cmd.FirstEnabledTube`	`int`	Returns the index of the first tube to be switched, i.e. TubeIsOn[FirstEnabledTube]=true.

Example - `OnCoolant` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnCoolant(ICLDCoolantCommand cmd, CLDArray cld)
{
    if (cmd.IsOn) {
        nc.MCoolant.v = 8;
    } else {
        nc.MCoolant.v = 9;
        nc.Block.Out();
    }
}
```

See also

- CLData access model

CUTCOM - Tool compensation

Command caption

How the command appears in the CLData command list:

```
CUTCOM ON(71)|OFF(72), LENGTH(9)|R(23) lr, X x, Y y, Z z,
XY n, YZ m, XZ k, RIGHT(24)|LEFT(8)
```

Use CUTCOM command to switch compensation mode for length or radius of the tool.

Access from sppx

Handler: `program CutCom`

Parameter	CLD array	CLD name	Description
ON(71) OFF(72)	CLD[1]	CLD.OnOff	Command type: 71 – Enable compensation, 72 – Disable compensation.
LENGTH(9) R(23)	CLD[2]	CLD.Length	Compensation kind: 9 – tool length compensation, 23 – tool radius compensation.
LR	CLD[3]	CLD.LR	Tool length or radius compensation value.
x y z	CLD[4] CLD[5] CLD[6]	CLD.X CLD.Y CLD.Z	Compensation along axes.
n m k	CLD[7] CLD[8] CLD[9]	CLD.N CLD.M CLD.K	Plane compensation values.
RIGHT(24) LEFT(8)	CLD[10]	CLD.RGT	Tool radius compensation direction: 24 – right

Parameter	CLD array	CLD name	Description
			compensation, 8 – left compensation.

To create ISO postprocessor using masks use ISO.G values which correspond to CUTCOM parameters:

CLD parameter	ISO value
CLD[1] = 72, CLD[2] = 9	ISO.G = 49
CLD[1] = 71, CLD[2] = 9	ISO.G = 43
CLD[1] = 72, CLD[2] = 23	ISO.G = 40
CLD[1] = 71, CLD[2] = 23, CLD[10] = 24	ISO.G = 42
CLD[1] = 71, CLD[2] = 23, CLD[10] = 8	ISO.G = 41

Example (from the Fanuc (30i)_Mill postprocessor):

```
program CutCom
  if cld[2]=9 then begin                          ! LENGTH
    X = MaxReal; X@ = X
    Y = MaxReal; Y@ = Y
    Z = MaxReal; Z@ = Z
    if cld[1]=71 then begin ! On
      CoordSys = WCSNumber
      OutBlock
      if (NeedKorDl=0) and (OperationIsLathe=0) and
         ((WasLoadTool>0) or (H<>cld[3]))
      then begin
        NeedKorDl = 43
        H = cld[3]; H@ = H
      end
      OutBlock
    end else begin           ! Off
      KORDL = 49; !KORDL@ = MaxReal
      H = 0; H@ = H
      OutBlock

      call SwitchOffLocalCS
    end
  end else if cld[2]=23 then begin              ! RADIUS
    if cld[1]=72 then
```

```
      KorEcv = 40
    else begin
      if GFeed<>GFeed@ then OutBlock
      if cld[10]=24 then KorEcv=42 else KorEcv=41
      D = cld[3]
      D@ = MaxReal
    end
  end else begin
  end
end
```

Access from .NET

Handlers:

```
public override void OnCutCom(ICLDCutComCommand cmd, CLDArray cld)
public override void OnLengthCompensation(ICLDCutComCommand cmd, CLDArray cld)   //
LENGTH(9) compensation
public override void OnRadiusCompensation(ICLDCutComCommand cmd, CLDArray cld)   //
RADIUS(23) compensation
```

Inheritance: `ICLDCutComCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Returns "True" if it's a compensation switch ON command, otherwise returns "False".
`cmd.IsOff`	`bool`	Returns "True" if it's a compensation switch OFF command, otherwise returns "False".
`cmd.IsLength`	`bool`	Returns "True" if it's a LENGTH compensation switch command, otherwise returns "False".

Member	Type	Description
`cmd.IsRadius`	`bool`	Returns "True" if it's a RADIUS compensation switch command, otherwise returns "False".
`cmd.IsRightDirection`	`bool`	Defines direction for the compensation. "True" means right direction, "False" means left direction.
`cmd.IsLeftDirection`	`bool`	Defines direction for the compensation. "True" means left direction, "False" means right direction.
`cmd.CorrectorNumber`	`int`	Number of tool corrector in machine's correctors table.

Example - `OnLengthCompensation` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnLengthCompensation(ICLDCutComCommand cmd, CLDArray cld)
{
    if (cmd.IsOn) {
        nc.GLCompens.Reset(43);
        nc.HLCompens.Reset(cmd.CorrectorNumber);
    } else {
        nc.GLCompens.Show(49);
        nc.HLCompens.Hide(0);
        nc.Block.Out();
    }
}
```

Example - `OnRadiusCompensation` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnRadiusCompensation(ICLDCutComCommand cmd, CLDArray cld)
{
    if (cmd.IsOn) {
        if (cmd.IsLeftDirection)
            nc.GRCompens.Show(41);
        else
            nc.GRCompens.Show(42);
        nc.DRCompens.Show(cmd.CorrectorNumber);
    } else {
```

```
            nc.GRCompens.Show(40);
        }
    }
```

See also

- CLData access model

CYCLE - Hole machining cycles

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

Command caption

How the command appears in the CLData command list:

```
CYCLE ON(71) | OFF(72) | DRILL(163) | FACE(81) | DEEP(153) |
BRKCHP(288) | TAP(168) | BORE5(209) | BORE6(210) | BORE7(211) |
BORE8(212) | BORE9(213)
```

CYCLE command is used to pass standard drilling cycles G81-G89. Use CYCLE command to determine cycle type and process the cycle accordingly. Cycle type is specified by the first parameter, other parameters types and their number depends on the cycle type.

Access from sppx

Handler: `program Cycle`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.Type`), in .NET by the same name as a string key (e.g. `cld["Type"]`).

Parameter	CLD array	CLD name	Description
Type	CLD[1]	CLD.Type	Cycle type: 71(ON) – turn cycle on, 72(OFF) – turn cycle off, 163(DRILL) - G81 drilling cycle, 81(FACE) - G82 drilling cycle, 153(DEEP) - Deep drilling cycle, 288(BRKCHP) - Drilling with chip breaking, 168(TAP) - G84 Tapping cycle, 209(BORE5) -

Parameter	CLD array	CLD name	Description
			G85 drilling cycle, 210(BORE6) - G86 drilling cycle, 211(BORE7) - G87 drilling cycle, 212(BORE8) - G88 drilling cycle, 213(BORE9) - G89 drilling cycle.

Usually CAM system generates drilling technology similar to the following:

```
...
GOTO.abs X, Y, Z ! Position the tool at the first hole.
CYCLE DRILL(163) ! Perform the drilling calling cycle of appropriate type.
GOTO.abs X, Y, Z ! Move the tool to the next hole.
CYCLE DRILL(163) ! Perform the drilling calling cycle of appropriate type.
...
GOTO.abs X, Y, Z ! Move the tool to the last hole.
CYCLE DRILL(163) ! Perform drilling by calling cycle of appropriate type.
CYCLE OFF(72) ! Turn cycle off.
...
```

The first drilling cycle command should form the NC-block containing a cycle definition (like G81) and cycle parameters. For example:

```
G81 X_Y_Z_R_F_
```

Following commands processing should form block containing only the changed registers. For example:

```
X_Y_Z_
X_Y_Z_
...
X_Y_Z_
```

The `CYCLE OFF(72)` command should form the code of cancelling cycle. For example:

```
G80
```

When developing ISO postprocessors using masks ISO.G operator can be used. The value of ISO.G is set automatically according to the following table:

CLD parameter value	ISO.G value
CLD[1] = 163	ISO.G = 81
CLD[1] = 81	ISO.G = 82
CLD[1] = 168	ISO.G = 84
CLD[1] = 209	ISO.G = 85
CLD[1] = 210	ISO.G = 86
CLD[1] = 211	ISO.G = 87
CLD[1] = 212	ISO.G = 88
CLD[1] = 213	ISO.G = 89
CLD[1] = 153	ISO.G = 83
CLD[1] = 288	ISO.G = 73
CLD[1] = 72	ISO.G = 80

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Cycle
  if cld[1]=71 then CycleOn = 1 else ! cycle On
  if cld[1]=72 then CycleOn = 0 else ! cycle Off
  begin
    CycleOn = 1                      ! cycle On
    KodCycle = cld[1]
    Za = cld[2]                      ! work depth
    if cld[4] > 0 then begin         ! creating FEED_ value
      if cld[3]=316 then Fr=cld[4]*S else Fr=cld[4]
    end
    Zf = cld[5]                      ! safe level
    ZP_ = cld[6]                      ! reverse move level
    Dwell = cld[10]*1000              ! dwell time
    Zl = cld[6]                      ! depth of one drilling step
    Zi = cld[8]                    ! transitional value
```

```
  end

  if CycleOn=1 then begin
    !X@=maxreal; Y@ = maxreal;
    !Feed_@=Maxreal;
    Zcycle=Zf-Za
    ZClear=Zf
    p=0;  P@=0
    If KodCycle=153 or KodCycle=288 or KodCycle=211 then Q=Zl else Q=0
    If KodCycle=81 or KodCycle=213 then P=Dwell else P=0
    if KodCycle=163 then Cycle = 81   ! DRILL
    if KodCycle=81  then Cycle = 82   ! FACE
    if KodCycle=168 then begin
      Fr = S * cld[12]
      ! Klient: Pered cziklom narezaniya
      ! rezby dolzhna stoyat strochka M84S2000 (zhyostkoj)
      if (Cycle<>84) and (Cycle<>74) then begin
        if cld[14]=1 then  ! esli zhyostkij tip derzhatelya
          output "M84 S" + Str(S);
      end
      if Msp=4 then ! SHpindel nazad krutitsya, levaya relba
        Cycle = 74
      else          ! SHpindel vperyod krutitsya, obychnaya relba
        Cycle = 84;
    end;
    if KodCycle=209 then Cycle = 85   ! BORE5
    if KodCycle=210 then begin ! BORE6
      if cld[14]>0 then begin
        Cycle = 76;
        QStep = sqr(cld[6]^2+cld[7]^2);
!        DrillPause = cld[15]*1000;
!        if DrillPause=0 then
!          DrillPause@ = 0
      end else begin
        Cycle = 86;
      end
    end
    if KodCycle=211 then begin ! BORE7
      if cld[14]>0 then begin
        call Swap(ZCycle, ZClear)
        Cycle = 87;
        QStep = sqr(cld[6]^2+cld[7]^2);
!        DrillPause = cld[15]*1000;
!        if DrillPause=0 then
!          DrillPause@ = 0
      end else begin
```

```
        Cycle = 88;
      end
    end
    if KodCycle=212 then Cycle = 88   ! BORE8
    if KodCycle=213 then Cycle = 89   ! BORE9
    if KodCycle=153 then Cycle = 83   ! DEEP
    if KodCycle=288 then Cycle = 73   ! BRKCHP
    FEED_ = Fr
    OutBlock
  end else
    begin
      Cycle = 80;
      OutBlock;
    end
end
```

Access from .NET

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

Inheritance: `ICLDCycleCommand : ICLDCommand`. Read from the strongly-typed `cmd`, the raw `cld[i]` array (same indices as in sppx), or the named `cmd["..."]` helpers; the common `ICLDCommand` members are described in the CLData access model.

Example:

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
{
    // cld[1] - cycle type: 71 = ON, 72 = OFF, otherwise a specific cycle (DRILL
163, TAP 168, ...)
    if ((int)cld[1] == 72) {
        nc.G.Show(80);                  // cancel the canned cycle
        nc.Block.Out();
    } else if ((int)cld[1] != 71) {
        nc.G.Show((int)cld["ISO.G"]);  // ISO.G is the mapped G-code (G81..G89)
        nc.Z.Show(cld["H"]);           // hole bottom level
        nc.R.Show(cld["Top"]);         // retract level
        nc.F.Show(cld["F"]);           // feed
        nc.Block.Out();
    }
}
```

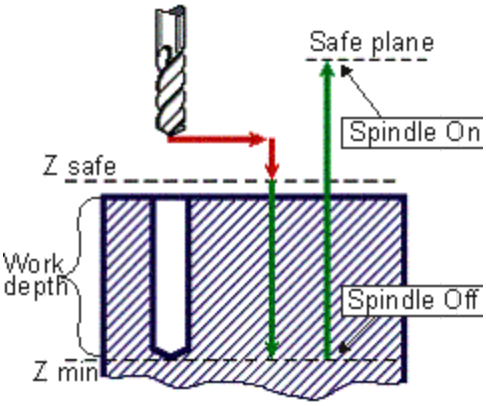
See also

- G81 drilling cycle `CYCLE_DRILL(163)`
- G82 drilling cycle `CYCLE_FACE(81)`
- G84 drilling cycle `CYCLE_TAP(168)`
- G85 drilling cycle `CYCLE_BORE5(209)`
- G86 drilling cycle `CYCLE_BORE6(210)`
- G87 drilling cycle `CYCLE_BORE7(211)`
- G88 drilling cycle `CYCLE_BORE8(212)`
- G89 drilling cycle `CYCLE_BORE9(213)`
- Deep drilling with chip removing `CYCLE_DEEP(153)`
- Deep drilling with chip breaking `CYCLE_BRKCHP(288)`
- CLData access model

CYCLE BORE5(209) - G85 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G85 boring cycle performs rapid approach to the safe level, work feedrate boring will spindle stop at Z min level and work feedrate retract to the safe plane level.



Cycle includes:

- Rapid approach to Z safe level.
- Work feedrate travel to Z min level.
- Spindle stop.
- Work feedrate retract to Safe plane level.
- Restore spindle rotation direction and speed.

Command

```
CYCLE BORE5(209), A a, MMPM(315), N nm, F f, P p, T t
```

or

```
CYCLE BORE5(209), A a, MMPR(316), M nm, F f, P p, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BORE5(209)	CLD[1]	CLD.Type	Cycle type 209 (BORE5).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6] CLD[7]		Reserved.
p	CLD[8]	CLD.P	Safe plane level.
	CLD[9] CLD[10]		Reserved.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd`/`cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

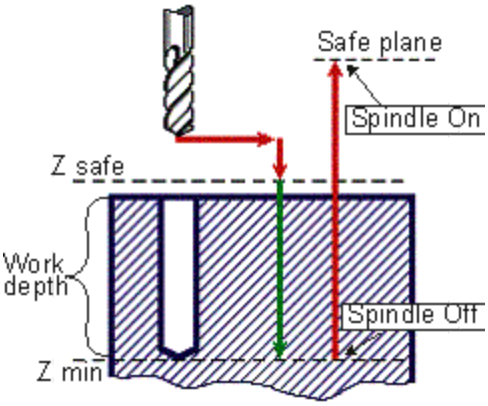
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE BORE6(210) - G86 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G86 boring canned cycle performs rapid approach to the safe level, work feedrate boring to the hole depth level, spindle stop at hole depth level and rapid retract to safe plane level.



G86 cycle includes:

- Rapid approach to the Z safe level.
- Work feedrate travel to the Z min level.
- Spindle stop. If oriented spindle stop is enabled (CLD [14] parameter), it is positioning the spindle in a given angular position and then tool shifts to specified small value of the X, Y and Z coordinates.
- Rapid retract to Safe plane level.
- Restore spindle rotation direction and speed.

Command

```
CYCLE BORE6(210), A a, MMPM(315), N nm, F f, P p, T t
```

or

```
CYCLE BORE6(210), A a, MMPR(316), M nm, F f, P p, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BORE6(210)	CLD[1]	CLD.Type	Cycle type 210 (BORE6)
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6]		The value of tool shift for the X coordinate after oriented spindle stop at the bottom of the hole.
	CLD[7]		The value of tool shift for the Y coordinate after oriented spindle stop at the bottom of the hole
p	CLD[8]	CLD.P	Safe plane level.
	CLD[9] CLD[10]		Reserved.
t	CLD[11]	CLD.Top	Hole top level.
	CLD[12]		The value of tool shift for the Z coordinate after oriented spindle

Parameter	CLD array	CLD name	Description
			stop at the bottom of the hole
	CLD[13]		The angular position of the spindle when stopping at the bottom of the hole (degrees)
	CLD[14]		The state of the oriented spindle stop at the bottom of the hole: 0 - off, 1 - enabled.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

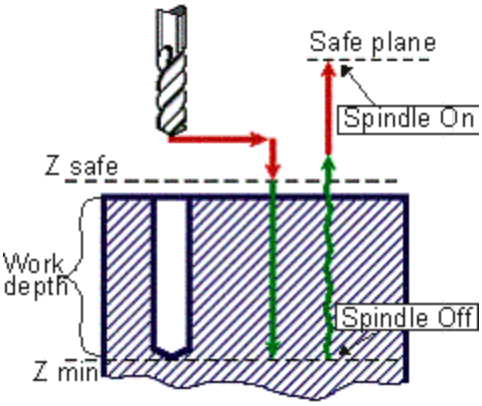
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE BORE7(211) - G87 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G87 boring canned cycle performs rapid approach to the safe level, work feedrate boring with spindle stop on hole depth level and manual retract to the safe plane level.



G87 cycle includes:

- Rapid approach to the Z safe level.
- Work feedrate travel to the Z min level.
- Spindle stop. If oriented spindle stop is enabled (CLD [14] parameter), it is positioning the spindle in a given angular position and then tool shifts to specified small value of the X, Y and Z coordinates.
- Manual retract to the Safe plane level.
- Restore spindle rotation direction and speed.

Command

```
CYCLE BORE7(211), A a, MMPM(315), N nm, F f, P p, T t
```

or

```
CYCLE BORE7(211), A a, MMPR(316), M nm, F f, P p, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BORE7(211)	CLD[1]	CLD.Type	Cycle type 211 (BORE7).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6]		The value of tool shift for the X coordinate after oriented spindle stop at the bottom of the hole.
	CLD[7]		The value of tool shift for the Y coordinate after oriented spindle stop at the bottom of the hole.
p	CLD[8]	CLD.P	Safe plane level.
	CLD[9] CLD[10]		Reserved.
t	CLD[11]	CLD.Top	Hole top level.
	CLD[12]		The value of tool shift for the Z coordinate after oriented spindle

Parameter	CLD array	CLD name	Description
			stop at the bottom of the hole.
	CLD[13]		The angular position of the spindle when stopping at the bottom of the hole (degrees)
	CLD[14]		The state of the oriented spindle stop at the bottom of the hole: 0 - off, 1 - enabled.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

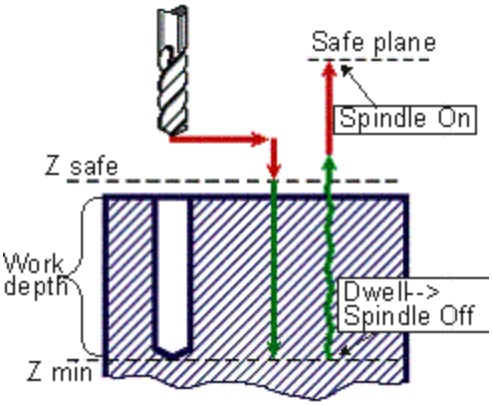
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE BORE8(212) - G88 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G88 boring canned cycle performs rapid approach to the safe level, work feedrate boring with dwell at the hole depth level, spindle stop at the hole depth level and manual retract to the safe plane level.



G88 Cycle includes:

- Rapid approach to the Z safe level.
- Work feedrate travel to the Z min level.
- Dwell.
- Spindle stop.
- Manual retract to the Safe plane level.
- Restore spindle rotation direction and speed.

Command

```
CYCLE BORE8(212), A a, MMPM(315), N nm, F f, P p, DWELL(279) h, T t
```

or

```
CYCLE BORE8(212), A a, MMPR(316), M nm, F f, P p, DWELL(279) h, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BORE8(212)	CLD[1]	CLD.Type	Cycle type 212 (BORE8).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6] CLD[7]		Reserved.
p	CLD[8]	CLD.P	Safe plane level.
DWELL	CLD[9]	CLD.DWELL	Dwell time keyword.
h	CLD[10]	CLD.H	Dwell time in seconds.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

See also

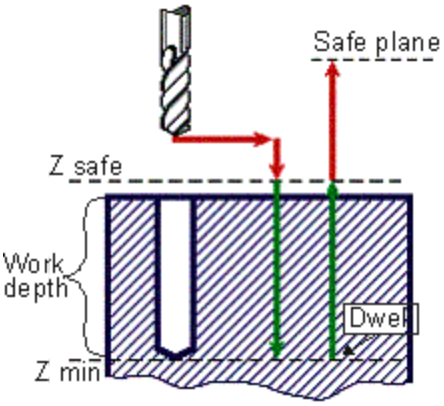
- Hole machining cycles (CYCLE)

- CLData access model

CYCLE BORE9(213) - G89 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G89 boring canned cycle performs rapid approach to the safe level, work feedrate boring with dwell at the hole depth level and work feedrate retract to the safe plane level.



G89 cycle includes:

- Rapid approach to the Z safe level.
- Work feedrate travel to the Z min level.
- Dwell.
- Work feedrate retract to the Safe plane level.

Command

```
CYCLE BORE9(213), A a, MMPM(315), N nm, F f, P p, DWELL(279) h, T t
```

or

```
CYCLE BORE9(213), A a, MMPR(316), M nm, F f, P p, DWELL(279) h, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BORE9(213)	CLD[1]	CLD.Type	Cycle type 213 (BORE9).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6] CLD[7]		Reserved.
p	CLD[8]	CLD.P	Safe plane level.
DWELL	CLD[9]	CLD.DWELL	Dwell time keyword.
h	CLD[10]	CLD.H	Dwell time in seconds.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

See also

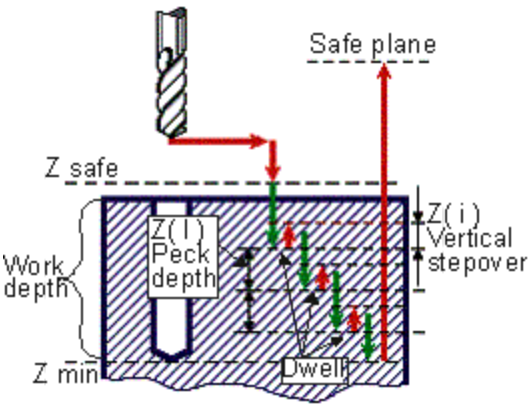
- Hole machining cycles (CYCLE)

- CLData access model

CYCLE BRKCHP(288) - Deep drilling with chip breaking

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

Deep drilling with chip removing cycle performs rapid approach to the safe level and repeated drilling according to specified cut-in depth value ZI and withdrawal Zi values.



Cycle includes:

- Rapid travel to the Z safe level.
- Work feedrate travel to the ZI.
- Rapid retract to the Zi withdrawal value and Dwell.
- Work feedrate travel to the ZI + Zi level.
- Rapid retract to the Zi withdrawal value and Dwell.
- Repeat until the hole depth level is reached.
- Rapid retract to the Safe plane level.

Command

```
CYCLE BRKCHP(288), A a, MMPM(315), N nm, F f,
L l, I i, P p, DWELL(279) h, T t
```

or

```
CYCLE BRKCHP(288), A a, MMPR(316), M nm, F f,
L l, I i, P p, DWELL(279) h, T t
```

Parameters

Parameter	CLD array	CLD name	Description
BRKCHP(288)	CLD[1]	CLD.Type	Cycle type 288 (BRKCHP).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
L	CLD[6]	CLD.L	Cut in value ZI
i	CLD[7]	CLD.I	Withdrawal value Zi
p	CLD[8]	CLD.P	Safe plane level.
DWELL	CLD[9]	CLD.DWELL	Dwell time keyword.
h	CLD[10]	CLD.H	Tool dwell in seconds.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

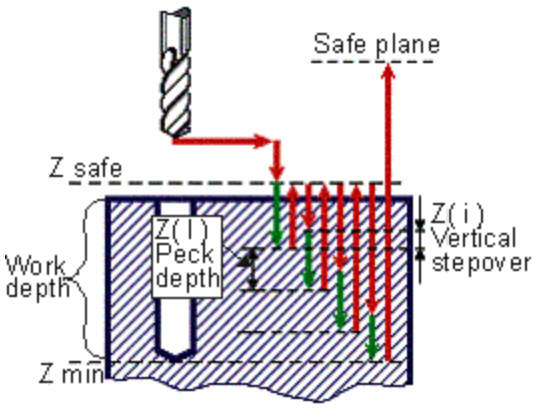
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE DEEP(153) - Deep drilling with chip removing

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

Deep drilling with chip removing cycle performs rapid approach to the safe level and repeated drilling according to specified cut-in depth value ZI and withdrawal Zi values.



Cycle includes:

- Rapid travel to the Z safe level.
- Work feedrate travel to the ZI.
- Rapid retract to the Z safe level.
- Rapid travel to the ZI - Zi level.
- Work feedrate travel to the ZI + Zi level.
- Rapid retract to the Z safe level.
- Rapid travel to the 2*ZI - Zi level.
- Work feedrate travel to the ZI + Zi level.
- Rapid retract to the Z safe level.
- Repeat until the hole depth level is reached.
- Rapid retract to the Safe plane level.

Command

```
CYCLE DEEP(153), A a, MMPM(315), N nm, F f, L l, I i, P p, T t
```

or

```
CYCLE DEEP(153), A a, MMPR(316), M nm, F f, L l, I i, P p, T t
```

Parameters

Parameter	CLD array	CLD name	Description
DEEP(153)	CLD[1]	CLD.Type	Cycle type 153 (DEEP).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Work feedrate value.
f	CLD[5]	CLD.F	Safe level.
L	CLD[6]	CLD.L	Cut in value Zl
i	CLD[7]	CLD.I	Withdrawal value Zi
p	CLD[8]	CLD.P	Safe plane level.
	CLD[9] CLD[10]		Reserved.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd`/`cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

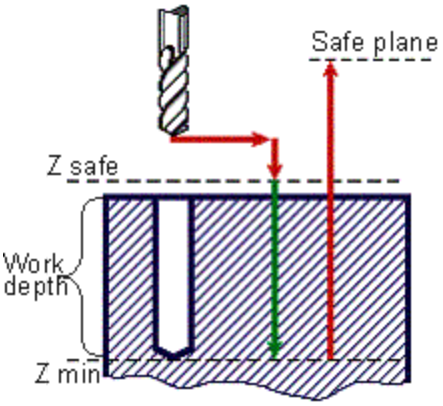
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE DRILL(163) - G81 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G81 simple drilling canned cycle performs drilling with rapid approach to the safe level and consequent retraction to the safe plane.



Cycle includes:

- Rapid travel to safe Z level.
- Work feedrate travel to Z min level.
- Rapid travel to Safe plane level.

Command

```
CYCLE DRILL(163), A a, MMPM(315), N nm, F f, P p, T t
```

or

```
CYCLE DRILL(163), A a, MMPR(316), M nm, F f, P p, T t
```

Parameters

Parameter	CLD array	CLD name	Description
DRILL(163)	CLD[1]	CLD.Type	Cycle type 163 (DRILL).

Parameter	CLD array	CLD name	Description
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Feedrate.
f	CLD[5] CLD[6], CLD[7]	CLD.F Reserved	Safe level.
p	CLD[8] CLD[9], CLD[10]	CLD.P Reserved	Safe plane level.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

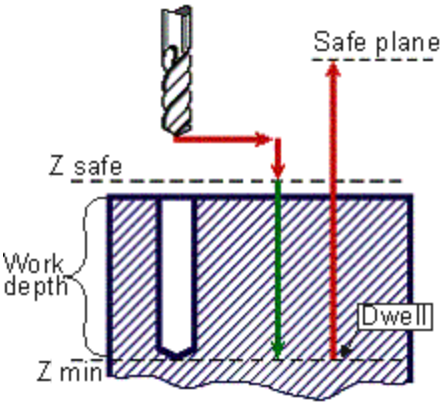
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE FACE(81) - G82 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G82 drilling with dwell canned cycle performs drilling with rapid approach to safe level, dwell at Z min level with work feed rate and retracts to the safe plane level at rapid feed rate. Dwell is specified by command parameters.



G82 drilling cycle includes:

- Rapid approach to Z safe level.
- Work feed rate travel to Z min level.
- Dwell at Z min level.
- Rapid retraction to Safe plane level.

Command

```
CYCLE FACE(81), A a, MMPM(315), N nm, F f, P p, DWELL(279) h, T t
```

or

```
CYCLE FACE(81), A a, MMPR(316), M nm, F f, P p, DWELL(279) h, T t
```

Parameters

Parameter	CLD array	CLD name	Description
FACE(81)	CLD[1]	CLD.Type	Cycle type 81 (FACE).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Feedrate value.
f	CLD[5]	CLD.F	Safe level.
	CLD[6] CLD[7]		Reserved.
p	CLD[8]	CLD.P	Safe plane level.
DWELL	CLD[9]	CLD.DWell	Dwell time keyword.
h	CLD[10]	CLD.H	Dwell time in seconds.
t	CLD[11]	CLD.Top	Hole top level.

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd/cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

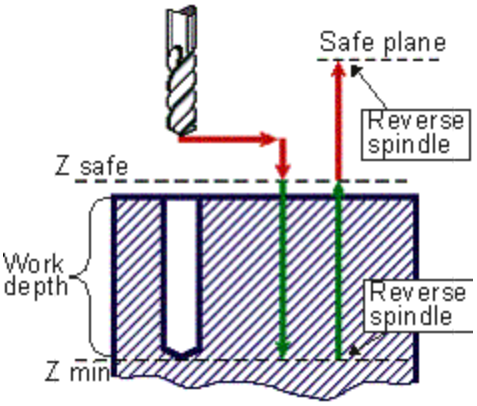
See also

- Hole machining cycles (CYCLE)
- CLData access model

CYCLE TAP(168) - G84 drilling cycle

> Kept for compatibility. CYCLE is retained only for compatibility with older postprocessors. Current versions of the CAM system generate the extended cycle command EXTCYCLE instead: it offers more capabilities and lets the CAM system fully simulate all of the cycle's nested motions, which CYCLE does not. Output of CYCLE can still be turned on in the CAM system settings, but this is not recommended.

G84 tapping canned cycle performs rapid approach to safe level, taps the thread on the feedrate retract with reverse spindle rotation.



G84 Cycle consist of:

- Rapid approach to Z safe level.
- Work feed rate travel to Z min level.
- Spindle rotation stop.
- Reverse spindle rotation and work feed rate return to Z safe level.
- Spindle rotation stop.
- Restore normal spindle rotation and rapid retract to Safe plane level.

Command

```
CYCLE TAP(168), A a, MMPM(315), N nm, F f,
P p, T t, S s, PS ps, SC sc, FR fr
```

or

```
CYCLE TAP(168), A a, MMPR(316), M nm, F f,
P p, T t, S s, PS ps, SC sc, FR fr
```

Parameters

Parameter	CLD array	CLD name	Description
TAP(168)	CLD[1]	CLD.Type	Cycle type 168 (TAP).
a	CLD[2]	CLD.A	Hole depth in current measure units (mm or inches).
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate measure units: 315 (MMPM) – mm per minute (inches per minute), 316 (MMPR) – mm per revolution (inches per revolution).
nm	CLD[4]	CLD.NM	Feedrate value
f	CLD[5]	CLD.F	Z safe level
L	CLD[6]	CLD.L	Cut in value Zl
i	CLD[7]	CLD.I	Withdrawal value Zi
p	CLD[8]	CLD.P	Safe plane level
DWELL(279)	CLD[9]	CLD.DWell	Dwell time keyword.
h	CLD[10]	CLD.H	Dwell time in seconds.
t	CLD[11]	CLD.Top	Hole top level
s	CLD[12]	CLD.Step	Threading pitch
ps	CLD[13]	CLD.Pos	Spindle start angle
sc	CLD[14]	CLD.Socket	Socket type: 0 – floating, 1 – fixed
fr	CLD[15]	CLD.FR	Return feedrate

Access from .NET

Delivered through the `OnCycle` handler (dispatch on the cycle type in `cmd`/`cld[1]`):

```
public override void OnCycle(ICLDCycleCommand cmd, CLDArray cld)
```

See Hole machining cycles (CYCLE) for the common .NET model.

See also

- Hole machining cycles (CYCLE)
- CLData access model

DELAY - Delay

Command caption

How the command appears in the CLData command list:

```
DELAY A a
```

DELAY command is used to pause program execution for the specified time. DELAY commonly translates to G04.

Access from sppx

Handler: `program Delay`

Parameter	CLD array	CLD name	Description
a	CLD[1]	CLD.A	Delay time in seconds

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Delay
  OutBlock
  GFunc = 4; GFunc@ = MaxReal
  Pause = CLD[1]; Pause@ = MaxReal
  OutBlock
end
```

Access from .NET

Handler:

```
public override void OnDelay(ICLDDelayCommand cmd, CLDArray cld)
```

Inheritance: `ICLDDelayCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.TimeSpan`	`double`	Delay time in seconds.

Example - `OnDelay` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnDelay(ICLDDelayCommand cmd, CLDArray cld)
{
    nc.GDelay.Show(4);
    nc.XDelay.Show(cmd.TimeSpan);
    nc.Block.Out();
}
```

See also

- CLData access model

EDMMOVE - EDM movement

Command caption

How the command appears in the CLData command list:

```
EDMMOVE MODE: RAPID(0) | 2AXIS(1) | TAPER(2) | MULTIPROF(3)| 4AXIS(4),
SPAN1: CUT(0) | ARC(1), SPAN2: CUT(0) | ARC(1),
EP1: (X x; Y y; Z z), EP2: (X x; Y y; Z z),
PC1: (X x; Y y), PC2: (X x; Y y), R1 r1, R2 r2, A a,
ROLLMODE: OFF(0) | SHARP(1) | CONICAL(2) | CYLINDRICAL(3) |
FIXED(4),
ROLLR1 rollr1, ROLLR2 rollr2
```

EDM movement command is generated by the Wire EDM operations of the CAM system. The command specifies the wire movement coordinates and sets the movement interpolation modes.

Access from sppx

Handler: `program EDMMove`

Parameter	CLD array	CLD name	Description
Movement mode	CLD[1]	CLD.Mode	RAPID(0) – Rapid movement, 2AXIS(1) – 2 axis movement, TAPER(2) – 2 axis movement with taper, MULTIPROF(3) – multiprofile movement (arcs and lines), 4AXIS(4) – 4 axis movement (lines only)
Lower contour element type	CLD[2]	CLD.Span1	CUT(0) – a line, ARC(1) – an arc
Upper contour element type	CLD[3]	CLD.Span2	CUT(0) – a line, ARC(1) – an arc

Parameter	CLD array	CLD name	Description
Lower contour element endpoint, EP1	CLD[4] CLD[5] CLD[6]	CLD.Ep1X CLD.Ep1Y CLD.Ep1Z	X, Y and Z coordinates of the lower contour element endpoint
Upper contour element endpoint, EP2	CLD[7] CLD[8] CLD[9]	CLD.Ep2X CLD.Ep2Y CLD.Ep2Z	X, Y and Z coordinates of the upper contour element endpoint
Lower contour arc center point, PC1	CLD[10] CLD[11]	CLD.Pc1X CLD.Pc1Y	X and Y coordinates of the lower contour arc center point, if this element is an arc.
Upper contour arc center point, PC2	CLD[12] CLD[13]	CLD.Pc2X CLD.Pc2Y	X and Y coordinates of the upper contour arc center point, if this element is an arc.
Lower contour arc radius, R1	CLD[14]	CLD.R1	If the lower contour element is an arc, then the parameter specifies the signed arc radius: R > 0 – the arc is counterclockwise, R < 0 – the arc is clockwise.
Upper contour arc radius, R2	CLD[15]	CLD.R2	If the upper contour element is an arc, then the parameter specifies the signed arc radius: R > 0 – the arc is counterclockwise, R < 0 – the arc is clockwise.
Taper angle, A	CLD[16]	CLD.A	In the TAPER(2) movement mode the parameter specifies signed taper angle: A >

Parameter	CLD array	CLD name	Description
			0 – right taper, A < 0 – left taper.
Sharp angle rolling mode, ROLLMODE	CLD[17]	CLD.RollMode	This parameter specifies sharp angle roll mode for the 2 axis and taper machining (CornerR function): OFF(0) – the corner rolling is off (R1 = 0, R2 = 0); SHARP(1) – in this mode only the lower contour angles are rolled (R1 = R1, R2 = 0); CONICAL(2) – both lower and upper contour angles are rolled, rounding radiuses are derived from taper angle (R1 = R1, R2 = R1 + Taper offset); CYLINDRICAL(3) – lower and upper contour angles are rounded, rounding radiuses are equal (R1 = R1, R2 = R1); FIXED(4) – lower and upper contour angles are rounded, rounding radiuses are independent (R1 = R1, R2 = R2)
Lower contour element roll radius, ROLLR1	CLD[18]	CLD.RollR1	This parameter specifies the rolling radius of the sharp angle of the lower contour element

Parameter	CLD array	CLD name	Description
Upper contour element roll radius, ROLLR2	CLD[19]	CLD.RollR2	This parameter specifies the rolling radius of the sharp angle of the upper contour element

Example (from the AgieCharmilles (ROBOFIL)_WireEDM postprocessor):

```
program EDMMove

  ! Set LCS
  if LCSSet<>1 then begin
    LCSSet = 1
    GCS = 92;         GCS@ = MaxReal
    X1  = XOrigin;    X1@  = MaxReal
    Y1  = YOrigin;    Y1@  = MaxReal
    ITop    = TopLevel;  ITop@    = MaxReal
    JBottom = BaseLevel; JBottom@ = MaxReal
    OutBlock
  end

  RollModeChanged = 0
  if (cld.RollMode<>0) and (RollOn<>48) then begin
    GRoll  = 48
    RollOn = 48
    RollModeChanged = 1
    GRoll@ = MaxReal
  end else
  if (cld.RollMode=0) and (RollOn<>49) then begin
    GRoll  = 49
    RollOn = 49
    GRoll@ = MaxReal
  end

  ! Insert and break wire commands
  if (cld.Mode<>0) and (WireInserted=0) and (MainProg=1) then begin
    call InsertWire
  end
  if (cld.Mode=0) and (WireInserted=1) and (MainProg=1) then begin
    if (cld.Ep1X<>Fp1X) or (cld.Ep1Y<>Fp1Y) then begin
      call BreakWire
    end
  end
```

```
  ! Output pass technology number
  if (cld.Mode<>0) and (WireInserted=1) then begin
    If OldSCode<>SCode then begin
      OldSCode  = SCode
      STechCode = SCode
      STechCode@ = MaxReal
      OUTBLOCK
    end
  end

  ! Start conical machining mode
  if (Cld.Mode=2) and (TaperMode<>60) then begin
    TaperMode  = 60
    TaperMode@ = MaxReal
    OUTBLOCK
  end

  ! Compensation turn on
  if (CompensNeedOut>0) and (GCompens<>40) then begin
    GCompens@ = MaxReal
    CompensNeedOut = 0
  end

  ! Motion mode turn on
  if cld.Mode=3 then G2Contour = 51 else ! 2Contour
  if cld.Mode=4 then GUV = 74            ! 4Axis_UV

!  if (G2Contour<>G2Contour@) or (GUV<>GUV@) then begin
!    OutBlock
!    GInterp1  = 0;
!    GInterp1@ = GInterp1;
!  end

  ! Compensation turn off
  if (CompensNeedOut>0) and (GCompens=40) then begin
    GCompens@ = MaxReal
    CompensNeedOut = 0
  end

  ! Coordinates output
  X1 = cld.Ep1X
  Y1 = cld.Ep1Y
  OUTG = 0
  case cld.Mode of
    0: begin ! Rapid
```

```
  if (Abs(X1-X1@)>Toler) or (Abs(Y1-Y1@)>Toler) then
    GInterp1 = 0 ! Rapid movement
end
1, 2, 3, 4: begin ! 2D Axis, Taper, 4D
  if (Abs(X1-X1@)>Toler) or (Abs(Y1-Y1@)>Toler) then begin
    if cld.Span1=1 then begin ! Arc
      if cld.R1>0 then GInterp1 = 3
                  else GInterp1 = 2
      I1 = cld.Pc1X - Fp1X;  I1@ = MaxReal
      J1 = cld.Pc1Y - Fp1Y;  J1@ = MaxReal
    end else begin ! Cut
        GInterp1 = 1
    end
    GInterp1@ = MaxReal
    OUTG = GInterp1
  end
  if cld.Mode=2 then begin      ! Taper
    if cld.A>0 then GTaper = 52
               else GTaper = 51
    A = abs(cld.A);
  end else
  if cld.Mode=3 then begin ! 4D
    U = cld.Ep2X; U@ = MaxReal
    V = cld.Ep2Y; V@ = MaxReal
    if cld.Span2=1 then begin ! Arc
      if cld.R2>0 then begin
        if (OutG=0) or (OutG<>3) then begin
          GInterp2 = 3
        end
      end else begin
        if (OutG=0) or (OutG<>2)  then begin
          GInterp2 = 2
        end
      end
      K = cld.Pc2X - Fp2X;  K@ = MaxReal
      L = cld.Pc2Y - Fp2Y;  L@ = MaxReal
    end else begin ! Cut
      if (Abs(U-U@)>Toler) or (Abs(V-V@)>Toler) then begin
        if (OutG=0) or (OutG<>1) then begin
          GInterp2 = 1
          GInterp2@ = MaxReal
        end
      end
    end
  end
  if ((cld.Mode=1) or (cld.Mode=2)) and
```

```
          (cld.RollMode>0)
        then begin  ! CornerR
          RollR1 = cld.RollR1;
          if RollModeChanged=1 then begin
            RollR1@ = MaxReal
            RollModeChanged =0
          end
        end
      end
    end

    ! Motion mode turn off
    if (cld.Mode<>2) and (GTaper<>50) then begin
      GTaper = 50
      !TaperMode = MaxReal;
    end
    if (cld.Mode<>4) and (GUV<>75) then begin
      U = 0; V = 0
    end
    OutBlock
    if (cld.Mode<>4) and (GUV<>75) then begin
      GUV = 75
      U = MaxReal; U@ = U
      V = MaxReal; V@ = V
    end
    OutBlock

    ! Remember current coordinates
    Fp1X = cld.Ep1X
    Fp1Y = cld.Ep1Y
    if cld.Mode<3 then begin ! 2D
      Fp2X = Fp1X
      Fp2Y = Fp1Y
    end else begin           ! 4D
      Fp2X = cld.Ep2X
      Fp2Y = cld.Ep2Y
    end
  end
```

Access from .NET

Handlers:

```
  public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
  group handler - before any motion command
```

```
public override void OnEDMMove(ICLDEDMMoveCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)   // group
handler - after any motion command
```

Inheritance: `ICLDEDMMoveCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.MotionMode`	`CLDEDMMotionMode`	Mode of the motion: Rapid, Plain2D, Taper2D, MultiProf4D or CutsOnly4D.
`cmd.IsRapidMove`	`bool`	Returns "True" if the motion mode is Rapid, otherwise returns "False".
`cmd.IsPlain2DMove`	`bool`	Returns "True" if the motion mode is Plain2D, otherwise returns "False".
`cmd.IsTaper2DMove`	`bool`	Returns "True" if the motion mode is Taper2D, otherwise returns "False".
`cmd.IsMultiProf4DMove`	`bool`	Returns "True" if the motion mode is MultiProf4D, otherwise returns "False".
`cmd.IsCutsOnly4DMove`	`bool`	Returns "True" if the motion mode is CutsOnly4D, otherwise returns "False".
`cmd.Lower`	`ICLDEDMMoveSpan`	Parameters of the lower guide motion (type, coordinates and radius).
`cmd.Upper`	`ICLDEDMMoveSpan`	Parameters of the upper guide motion (type, coordinates and

Member	Type	Description
		radius).
`cmd.TaperAngle`	`double`	In the Taper2D movement mode the parameter specifies signed taper angle: "A > 0" – right taper, "A < 0" – left taper.
`cmd.RollMode`	`EDMRollMode`	Sharp corners rounding mode for the 2 axis and taper machining (CornerR function, off, sharp, conical, cylindrical or fixed).
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement

Example - `OnEDMMove` (from the AgieCharmilles_WireEDM_DN postprocessor):

```
public override void OnEDMMove(ICLDEDMMoveCommand cmd, CLDArray cld)
{
    // Insert and break wire commands
    if (!cmd.IsRapidMove && !wireInserted)
    {
        InsertWire();
    }
    if (cmd.IsRapidMove && wireInserted)
    {
        if (cmd.Lower.EP != previousLowerPoint)
            BreakWire();
    }

    // Cutting conditions selection
    if (conditionsNeedOut)
    {
        nc.C.Show();
        nc.Block.Out();
        conditionsNeedOut = false;
    }

    // Compensation turn on
    if (compensNeedOut && nc.GCompens != 40)
    {
        nc.GCompens.Show();
```

```
    nc.H.Show();
    nc.Block.Out();
    compensNeedOut = false;
}

// Motion mode turn on
if (cmd.IsMultiProf4DMove)
{
    nc.G2Contour.v = 61;
}
else if (cmd.IsCutsOnly4DMove)
{
    nc.GUV.v = 74;
}
if (nc.G2Contour.ValuesDiffer || nc.GUV.ValuesDiffer)
{
    nc.Block.Out();
    nc.GInterp1.RestoreDefaultValue(false);
    nc.GInterp2.RestoreDefaultValue(false);
}

// Compensation turn off
if (compensNeedOut && nc.GCompens == 40)
{
    nc.GCompens.Show();
    compensNeedOut = false;
}

// Coordinates output
nc.X1.v = cmd.Lower.EP.X;
nc.Y1.v = cmd.Lower.EP.Y;
nc.Z1.v = cmd.Lower.EP.Z;
switch (cmd.MotionMode)
{
    case CLDEDMMotionMode.Rapid:
        if (nc.X1.ValuesDiffer || nc.Y1.ValuesDiffer || nc.Z1.ValuesDiffer)
        {
            nc.GInterp1.v = 0;
        }
        break;
    case CLDEDMMotionMode.Plain2D:
    case CLDEDMMotionMode.Taper2D:
    case CLDEDMMotionMode.MultiProf4D:
        if (cmd.Lower.IsArc)
        {
            nc.GInterp1.v = cmd.Lower.ArcR > 0d ? 3 : 2;
```

```
                nc.I1.Show(cmd.Lower.Center.X - previousLowerPoint.X);
                nc.J1.Show(cmd.Lower.Center.Y - previousLowerPoint.Y);
            }
            else // Cut
            {
                nc.GInterp1.v = 1;
            }
            if (cmd.IsTaper2DMove)
            {
                nc.GTaper.v = cmd.TaperAngle > 0 ? 52 : 51;
                nc.A.Show(Abs(cmd.TaperAngle));
            }
            else if (cmd.IsMultiProf4DMove)
            {
                nc.Colon.Show();
                nc.X2.v = cmd.Upper.EP.X;
                nc.Y2.v = cmd.Upper.EP.Y;
                nc.Z2.v = cmd.Upper.EP.Z;
                if (cmd.Upper.IsArc)
                {
                    nc.GInterp2.v = cmd.Upper.ArcR > 0d ? 3 : 2;
                    nc.I2.Show(cmd.Upper.Center.X - previousUpperPoint.X);
                    nc.J2.Show(cmd.Upper.Center.Y - previousUpperPoint.Y);
                }
                else // Cut
                {
                    nc.GInterp2.v = 1;
                }
                // CornerR
                if ((cmd.IsPlain2DMove || cmd.IsTaper2DMove) && cmd.RollMode
!= EDMRollMode.Off)
                {
                    nc.RollR1.Show(cmd.Lower.RollR);
                    nc.RollR2.Show(cmd.Upper.RollR);
                }
            }
            break;
        case CLDEDMMotionMode.CutsOnly4D:
            nc.GInterp1.v = 1;
            var d = (T3DPoint)cmd.Upper.EP - cmd.Lower.EP;
            nc.U.v = d.X;
            nc.V.v = d.Y;
            nc.W.v = d.Z;
            break;
    }
```

```
    // Motion mode turn off
    if (!cmd.IsTaper2DMove && nc.GTaper != 50)
    {
        nc.GTaper.v = 50;
    }
    if (!cmd.IsMultiProf4DMove && nc.G2Contour != 60)
    {
        nc.G2Contour.v = 60;
    }
    if (!cmd.IsCutsOnly4DMove && nc.GUV != 75)
    {
        nc.U.v = 0;
        nc.V.v = 0;
    }
    nc.Block.Out();
    if (!cmd.IsCutsOnly4DMove && nc.GUV != 75)
    {
        nc.GUV.v = 75;
        nc.U.Hide();
        nc.V.Hide();
    }
    nc.Block.Out();

    // Remember current coordinates
    previousLowerPoint = cmd.Lower.EP;
    if (!cmd.IsMultiProf4DMove && !cmd.IsCutsOnly4DMove) // 2D
    {
        previousUpperPoint = previousLowerPoint;
    }
    else // 4D
    {
        previousUpperPoint = cmd.Upper.EP;
    }
}
```

See also

- CLData access model

EFFECTOR - End effector activation

Command caption

How the command appears in the CLData command list:

```
EFFECTOR N n, Value ON(71) | OFF(72)
```

EFFECTOR CLData command should activate or deactivate the end effector of a machine or robot (laser, jet nozzle, arc welder etc.).

Access from sppx

Handler: `program Effector`

Parameter	CLD array	CLD name	Description
N	CLD[1]	CLD.N	Unique identifier of the end effector to be enabled or disabled
Value	CLD[2]	CLD.Value	Type of the command: 71 - switch ON, 72 - switch OFF.

Example:

```
program Effector
  ! cld[1] - effector id, cld[2] - 71 (ON) | 72 (OFF)
  if cld[2] = 71 then
    Output "DOUT M#(" + cld[1] + ")=ON"
  else
    Output "DOUT M#(" + cld[1] + ")=OFF"
end
```

Access from .NET

Handler:

```
public override void OnEffector(ICLDEffectorCommand cmd, CLDArray cld)
```

Inheritance: `ICLDEffectorCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Returns "true" if this command is EFFECTOR switch ON command, otherwise "false".
`cmd.IsOff`	`bool`	Returns "true" if this command is EFFECTOR switch OFF command, otherwise "false".
`cmd.EffectorID`	`int`	Numerical identifier of the effector to activate-deactivate.

Example - `OnEffector` (from the CRP_robot_DN postprocessor):

```
public override void OnEffector(ICLDEffectorCommand cmd, CLDArray cld)
{
//    if (cmd.IsOn && !effectorIsOn) {
//        prg.WriteLine("DOUT M#(111)=ON");
//    } else if (cmd.IsOff  && effectorIsOn) {
//        prg.WriteLine("DOUT M#(111)=OFF");
//    }
    effectorIsOn = cmd.IsOn;
}
```

See also

- CLData access model

EXTCYCLE - Extended cycle

Command caption

How the command appears in the CLData command list:

```
EXTCYCLE ON(71)|CALL(52)|OFF(72)
DRILL(163)|FACE(81)|TAP(168)|DEEP(153)|W5DDrill(481)|
         W5DBore5(485)|LATHEROUGH(401)|LATHETHREAD(403)|... {, a} {, b} {, c} ...
```

Use the extended cycle command EXTCYCLE to process standard cycles. The following is the list of standard cycles supported by the EXTCYCLE command:

- Lathe finish cycle G70, G73 - `LATHEFINISH(400)`
- Lathe roughing cycle G71, G72 - `LATHEROUGH(401)`
- Lathe grooving cycle G74,G75 - `LATHEGROOVE(402)`
- Lathe threading cycle G76 - `LATHETHREAD(403)`
- Lathe threading single pass cycle G92 - `LATHETHREADG92(404)`
- Lathe drilling cycle G81 - `DRILL(163)`
- Lathe drilling cycle G82 - `FACE(81)`
- Lathe tapping cycle G84 - `TAP(168)`
- Lathe drilling with chip breaking cycle - `BRKCHP(288)`
- Lathe drilling with chip removing cycle - `DEEP(153)`
- Drilling cycle G81 - `W5DDrill(481)`
- Drilling cycle G82 - `W5DFace(482)`
- Drilling with chip removing cycle G83 - `W5DChipRemoving(483)`
- Drilling with chip breaking cycle G73 - `W5DChipBreaking(473)`
- Tapping cycle G84 - `W5DTap(484)`
- Drilling cycle G85 - `W5DBore5(485)`
- Drilling cycle G86 - `W5DBore6(486)`
- Drilling cycle G87 - `W5DBore7(487)`
- Drilling cycle G88 - `W5DBore8(488)`
- Drilling cycle G89 - `W5DBore9(489)`
- Thread milling cycle - `W5DThreadMill(490)`
- Hole pocketing cycle - `W5DHolePocketing(491)`
- Probing cycle - `WProbing(500)`

Access from sppx

Handler: `program ExtCycle`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
ON(71) CALL(52) OFF(72)	CLD[1]	CLD.SubCmd	Command modifier: ON(71) – turn standard cycle ON, CALL(52) – call standard cycle, OFF(72) – turn standard cycle OFF.
LATHEFINISH(400) LATHEROUGH(401) LATHEGROOVE(402) LATHETHREAD(403) DRILL(163) FACE(81) TAP(168) BRKCHP(288) DEEP(153) W5DDrill(481) W5DFace(482) W5DChipRemoving(483) W5DChipBreaking(473) W5DTap(484) W5DBore5(485) W5DBore6(486) W5DBore7(487) W5DBore8(488) W5DBore9(489) W5DThreadMill(490) W5DHolePocketing(491) WProbing(500)	CLD[2]	CLD.SubType	Cycle type identifier.
a, b, c, d …	CLD[3] – CLD[258]	CLD.CLParams(1) – CLD.CLParams(256)	Cycle parameters list.

The first parameter is used to specify the start, call and cancellation of the standard cycle.

CAM system usually generates technology command sequence containing EXTCYCLE command similar to the following:

```
EXTCYCLE ON(71) ...
FEDRAT ...
GOTO ...
GOTO ...
EXTCYCLE CALL(52) ...
FEDRAT ...
GOTO ...
GOTO ...
EXTCYCLE CALL(52) ...
FEDRAT ...
GOTO ...
GOTO ...
EXTCYCLE CALL(52) ...
EXTCYCLE OFF(72) ...
```

The list of parameters depends on the cycle type.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program ExtCycle
  if cld[1] = 71 then CycleOn = 1                    ! cycle On
  else if cld[1] = 72 then begin                     ! cycle Off
    if (CLD.SubType >= 473) and (CLD.SubType <= 489) then Cycle = 80
    OutBlock
  end else if cld[1] = 52 then begin                 ! cycle Call
    if cld[10] > 0 then
      if cld[9] = 0 then Fr = cld[10]*S else Fr = cld[10]   ! feed units / value
  end
```

Access from .NET

Besides the base `OnExtCycle`, the command is dispatched (by cycle family) to one of the specialized handlers below:

```
public override void OnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
// base
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)    //
hole cycles
// also, by cycle family: OnTurnExtCycle, OnProbeExtCycle,
OnWeldingExtCycle, OnPickAndPlaceExtCycle
```

Inheritance: `ICLDExtCycleCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model. The raw **`cld[i]`** array is available with the same indices as in sppx, and the parameters are also reachable by numerical code through `cmd.Prm`.

Member	Type	Description

`cmd.IsOn`
`cmd.IsOff`
`cmd.IsCall`
	bool	True if it is a cycle switch-ON / switch-OFF / CALL command.

`cmd.CycleType`	int	Numerical code of the cycle type (simple drilling, deep drilling, tapping, OD turning, ...). See the constants in the `CLDCycle` class.
`cmd.Prm`	`ICodeParametersOfCLDCommand`	The parameter list; get a value by its numerical code.

`cmd.Prm` gives `ParamCount`, `Code(i)`, `ParamType(i)`, `IndexOfCode(code)`, and value indexers `Prm.Flt[code]` / `Prm.Int[code]` / `Prm.Bol[code]` / `Prm.Str[code]`.

Example (from the Sinumerik 840D mill postprocessor - note the `cld[i]` access):

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
{
    if (cmd.IsOn)
    {
        nc.GFeed.v = cld[9] == 1 ? 94 : 95;   // work-feed units (same cld index
as sppx)
        nc.Feed.v = cld[10];                  // work-feed value
        nc.Block.Out();
    }
    else if (cmd.IsOff)
        nc.G.Show(80);                        // cancel the cycle
    // else if (cmd.IsCall) dispatch by cmd.CycleType ...
}
```

See also

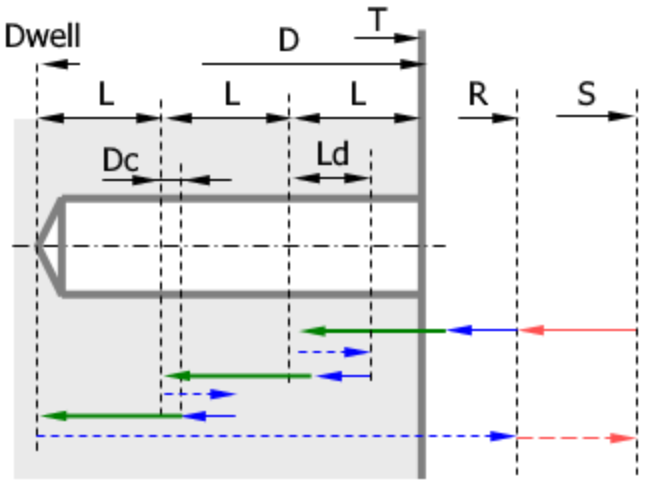
- CLData access model

EXTCYCLE BRKCHP - Lathe drilling with chip breaking cycle

Deep drilling cycle with tool returns for chip breaking `BRKCHP(288)` is used to machine axial holes with static tool in the spindle rotated workpiece. The cycle consists of:

1. Rapid approach to the axial safe plane level S.
2. Travel to the return level R.
3. Travel to the level of previous drill (equals the hole top level for the first drill) with Decelation (Dc).
4. Work feedrate motion to the length L distance.
5. Dwell for the specified time.
6. Tool retraction to the specified retraction distance Ld.
7. Repeat of steps 3 to 6 until the full hole depth is reached.
8. Return to the safe level R.
9. Rapid return to safe plane level S.

Drill step is decreased at each iteration for the specified value if decrease value is not zero.



Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: BRKCHP(288).

CLD array	CLD name	Description
CLD[3]	CLD.CLParams(1)	Upper level of hole (T).
CLD[4]	CLD.CLParams(2)	Hole depth from upper level (D).
CLD[5]	CLD.CLParams(3)	Return level (R).
CLD[6]	CLD.CLParams(4)	Safe level (S).
CLD[7]	CLD.CLParams(5)	Delay at low level in seconds (Dwell).
CLD[8]	CLD.CLParams(6)	Feed rate type: 0 – mm/rev, 1 – mm/min.
CLD[9]	CLD.CLParams(7)	Work feed.
CLD[10]	CLD.CLParams(8)	Return feed.
CLD[11]	CLD.CLParams(9)	Step for breaking bore chip (L)
CLD[12]	CLD.CLParams(10)	Deceleration (Dc)
CLD[13]	CLD.CLParams(11)	Return for breaking (Ld)
CLD[14]	CLD.CLParams(12)	Decrease step L for next iteration

Access from .NET

This cycle has no dedicated family handler - it is delivered through the base `OnExtCycle` (dispatch on `cmd.CycleType` / `cld[2]`). See Extended cycle (EXTCYCLE).

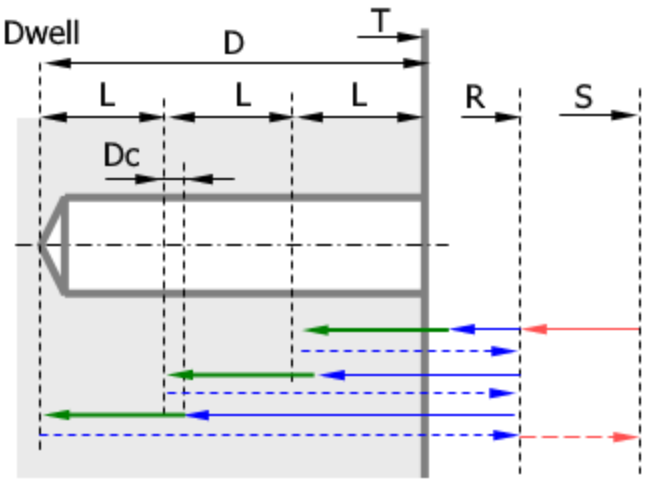
See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE DEEP - Lathe drilling with chip removing cycle

Deep drilling cycle with tool retraction for chip removing `DEEP(153)` is used to machine axial holes by a static tool. The cycle includes:

1. Rapid motion to safe plane level S.
2. Travel to the return level R.
3. Travel to the previous iteration depth (equals the hole top level for the first iteration) with the deceleration Dc.
4. Work feedrate motion to the length L distance.
5. Dwell for the specified time.
6. Return to the return level R.
7. Repeat steps 3 - 6 until the full hole depth is reached.
8. Rapid retract to the safe plane level S.



Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: DEEP(153).
CLD[3]	CLD.CLParams(1)	Upper level of hole (T).

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	Hole depth from upper level (D).
CLD[5]	CLD.CLParams(3)	Return level (R).
CLD[6]	CLD.CLParams(4)	Safe level (S).
CLD[7]	CLD.CLParams(5)	Delay at low level in seconds (Dwell).
CLD[8]	CLD.CLParams(6)	Feed rate type: 0 – mm/rev, 1 – mm/min.
CLD[9]	CLD.CLParams(7)	Work feed.
CLD[10]	CLD.CLParams(8)	Return feed.
CLD[11]	CLD.CLParams(9)	Step for removing bore chip (L)
CLD[12]	CLD.CLParams(10)	Deceleration (Dc)
CLD[13]	CLD.CLParams(11)	Reserved
CLD[14]	CLD.CLParams(12)	Decrease step L for next iteration

Access from .NET

This cycle has no dedicated family handler - it is delivered through the base `OnExtCycle` (dispatch on `cmd.CycleType` / `cld[2]`). See Extended cycle (EXTCYCLE).

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE DRILL - Lathe drilling cycle (G81)

Lathe drilling canned cycle `DRILL(163)` G81 drills axial holes with static tool inside rotated workpiece. The tool approaches the hole at the safe plane level descends at rapid feedrate to the return level, machines the hole at work feedrate, and rapidly retracts to the safe plane level.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: DRILL(163).
CLD[3]	CLD.CLParams(1)	The top level of a hole (T).
CLD[4]	CLD.CLParams(2)	The hole depth measured relative to the top level (D).
CLD[5]	CLD.CLParams(3)	The return level (R).
CLD[6]	CLD.CLParams(4)	The Safe level (S).
CLD[7]	CLD.CLParams(5)	Reserved.
CLD[8]	CLD.CLParams(6)	The federate units: 0 – mm/rev, 1 – mm/min.
CLD[9]	CLD.CLParams(7)	The Work feed.
CLD[10]	CLD.CLParams(8)	The Retract feed.

Access from .NET

This cycle has no dedicated family handler - it is delivered through the base `OnExtCycle` (dispatch on `cmd.CycleType` / `cld[2]`). See Extended cycle (EXTCYCLE).

See also

- Extended cycle (EXTCYCLE)

- CLData access model

EXTCYCLE FACE - Lathe drilling cycle (G82)

Lathe drilling canned cycle `FACE(81)` G82 drills axial holes with static tool inside rotated workpiece. The tool approaches the hole at the safe plane level descends at rapid feedrate to the return level, machines the hole at work feedrate, dwells at the hole bottom, and rapidly retracts to the safe plane level.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: FACE(81).
CLD[3]	CLD.CLParams(1)	The top level of a hole (T).
CLD[4]	CLD.CLParams(2)	The hole depth measured relative to the top level (D).
CLD[5]	CLD.CLParams(3)	The return level (R).
CLD[6]	CLD.CLParams(4)	The Safe level (S).
CLD[7]	CLD.CLParams(5)	Dwell at hole bottom level (seconds).
CLD[8]	CLD.CLParams(6)	The federate units: 0 – mm/rev, 1 – mm/min.
CLD[9]	CLD.CLParams(7)	The Work feed.
CLD[10]	CLD.CLParams(8)	The Retract feed.

Access from .NET

This cycle has no dedicated family handler - it is delivered through the base `OnExtCycle` (dispatch on `cmd.CycleType` / `cld[2]`). See Extended cycle (EXTCYCLE).

See also

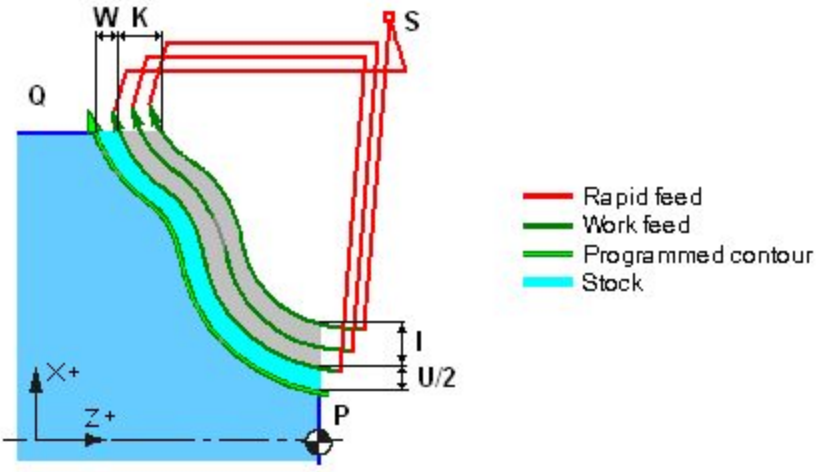
- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE LATHEFINISH - Lathe finish cycle (G70, G73)

`LATHEFINISH(400)` (G73) standard cycle is used for the rough machining of prior formed (molded, for example) workpiece. When machining with the standard cycle it is assumed that the material is removed or missing at a given distance from the programmed tool toolpath (PQ-block).

When cycle machining the values of F, S and T defined before the G73 frame or directly in the G73 frame. Feedrate (F), spindle speed (S) and tool change (T) commands between P and Q lines are ignored.

The first roughing cut offset is defined as (U/2 + I) along the X axis and (W + K) along the Z axis. Every following roughing cut approaches the last roughing cut contour by (I/(D-1)) along the X axis and (K/(D-1)) along the Z axis. The last roughing cut always leaves stock material U/2 for the X axis and W for the Z axis.



LATHEFINISH canned cycle is also used to form lathe finishing cycle G70 along the contour. If stocks are zero and there is only one cut then the lathe finishing G70 cycle is called.

Lathe cycles assume the part programmed contour PQ is defined. CAM system passes the contour as NC-subroutine. The number of the subroutine is specified in the `CLD[3]` parameter.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) — canned cycle on, CALL(52) — canned cycle call, OFF(72) — canned cycle off.

CLD array	CLD name	Description
CLD[2]	CLD.Subtype	Canned cycle type identifier: LATHEFINISH(400)
CLD[3]	CLD.CLParams(1)	The number of the NC subprogram that contains the machined contour PQ.
CLD[4]	CLD.CLParams(2)	The pass count (D). If one pass required then D=1, and occur G70. If D>1, then occur G73 cycle.
CLD[5]	CLD.CLParams(3)	The signed thickness of layer directed in Z axis Z (K). It is equal to zero for G70.
CLD[6]	CLD.CLParams(4)	The signed thickness by X (I) axis. It is equal to zero for G70 cycle.
CLD[7]	CLD.CLParams(5)	The signed stock by Z (W) axis. It is equal to zero for G70 cycle.
CLD[8]	CLD.CLParams(6)	The signed stock by X (U/2) axis. It is equal to zero for G70 cycle.
CLD[11]	CLD.CLParams(9)	Machining type: 0 — Face, 1 — OD (external), 2 — ID (internal). "Face" is a very rare case. Occurs only if the contour does not change along the axial coordinate with absolute accuracy. Most often, even in the case of face turning, there is a non-zero retract along the axis, which turns the face turning into an external (OD) or internal (ID) one.

CLD array	CLD name	Description
CLD[12]	CLD.CLParams(10)	Availability of finish pass (G70): 0 — without finish pass, 1 — with finish pass.
CLD[13]	CLD.CLParams(11)	Equidistant stock for contour

If a "signed" parameter is positive it is laid off in the positive direction of the respective axis, otherwise it is laid off in the negative direction.

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the turning-family handler `OnTurnExtCycle`:

```
public override void OnTurnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

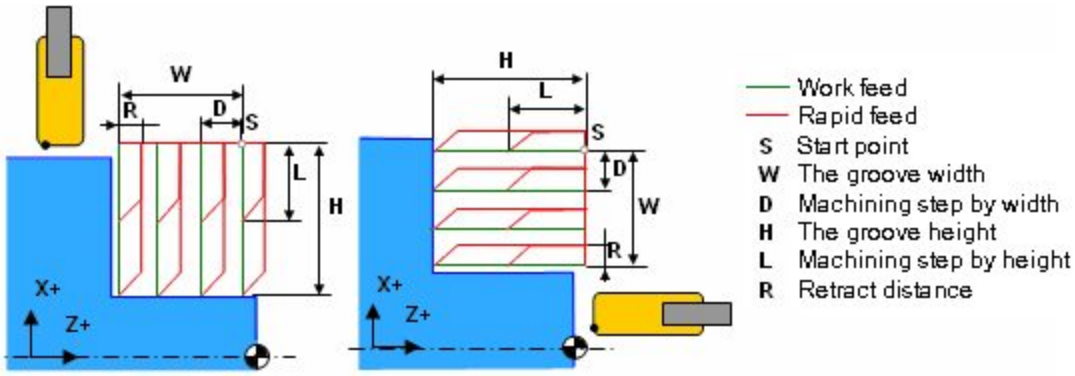
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE LATHEGROOVE - Lathe grooving cycle (G74, G75)

Cycle `LATHEGROOVE(402)` implements the G74 cylindrical grooving canned cycle and G75 face grooving canned cycle. Groove shape is rectangular and is defined by its dimensions: width and height. The cutting is done by consequent work passes of the tool with Z and X axes stepping.



Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: LATHEGROOVE(402)
CLD[3]	CLD.CLParams(1)	The groove type: 0 – turn groove (G75), 1 – face groove (G74)
CLD[4]	CLD.CLParams(2)	The signed width of the groove measured relative to the start point (W)
CLD[5]	CLD.CLParams(3)	The signed height of the groove measured relative to the start point (H)
CLD[6]	CLD.CLParams(4)	The absolute amount of the machining step by width (D)

CLD array	CLD name	Description
CLD[7]	CLD.CLParams(5)	The absolute value of the machining step by height (chip breaking depth, L)
CLD[8]	CLD.CLParams(6)	The absolute distance of the side retract before return of tool (R)
CLD[9]	CLD.CLParams(7)	The absolute value of the lead out for chip breaking.
CLD[10]	CLD.CLParams(8)	Delay at the bottom in seconds
CLD[11]	CLD.CLParams(9)	Chip breaking mode: 0 - do not use, 1 - on first plunge, 2 - on every plunge.
CLD[12]	CLD.CLParams(10)	Width rough stock
CLD[13]	CLD.CLParams(11)	Height rough stock

If a "signed" parameter is positive it is laid off in the positive direction of respective axis, otherwise it is laid off in negative direction.

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the turning-family handler `OnTurnExtCycle`:

```
public override void OnTurnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

See Extended cycle (EXTCYCLE) for the common .NET model.

See also

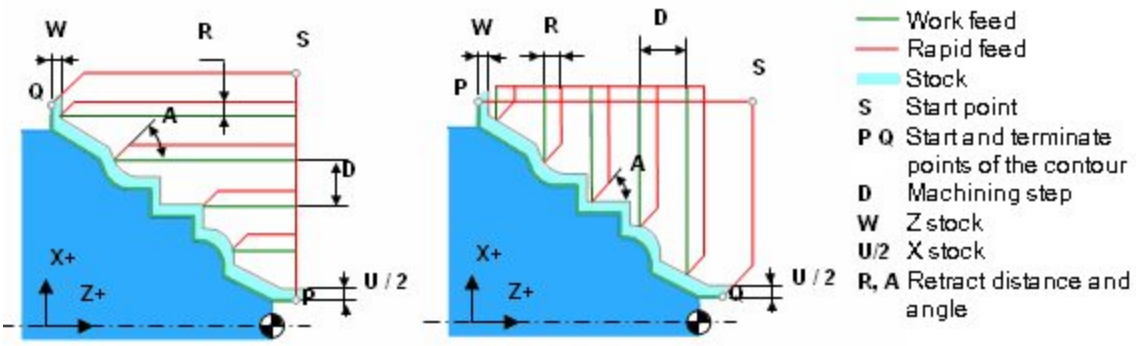
- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE LATHEROUGH - Lathe roughing cycle (G71, G72)

`LATHEROUGH(401)` canned cycle G71 performs roughing machining of the workpiece by the given part shape. F, S and T commands in the G71 line or already set before G71 line are used in the lathe roughing cycle G71. G71 cycle handles two machining toolpath types. If in the programmed toolpath the X axis doesn't change it's direction, then it's the first type toolpath (type I). The second toolpath type (type II) allows the change of the X axis direction. The Z axis direction must not change for both types.

G73 canned cycle is similar to G71, but is used to machine part face.

The last motion for both cycles is the return to the starting S position. As shown on the image S is the tool position when the cycle was called.



Lathe cycles assume the part programmed contour PQ is defined. CAM system passes the contour as NC-subroutine. The number of the subroutine is specified in the `CLD[3]` parameter.

If the parameter CLD[12]=1, then after roughing cycle G71 or G72 can be caused by a finishing pass along the contour - G70 cycle.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: LATHEROUGH(401)
CLD[3]	CLD.CLParams(1)	The number of the subroutine that machines the PQ contour.

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	The step of machining (D).
CLD[5]	CLD.CLParams(3)	The direction of machining: 0 – along the turn axis (G71), 1 – perpendicular to the turn axis (G72).
CLD[6]	CLD.CLParams(4)	The overlap mode switch: 0 – the overlap option is off, 1 – the overlap option is on (the toolpath contains additional tool movements along the machined contour cutting scallops)
CLD[7]	CLD.CLParams(5)	The signed Z (W) stock
CLD[8]	CLD.CLParams(6)	The signed X(U/2) stock
CLD[9]	CLD.CLParams(7)	The retract distance (R)
CLD[10]	CLD.CLParams(8)	The retract angle (A)
CLD[11]	CLD.CLParams(9)	Machining type: 1 – OD, 2 – ID
CLD[12]	CLD.CLParams(10)	Availability of finish pass (G70): 0 - without finish pass, 1 - with finish pass.
CLD[13]	CLD.CLParams(11)	Equidistant stock for contour

If a "signed" parameter is positive it is laid off in the positive direction of respective axis, otherwise it is laid off in negative direction.

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the turning-family handler `OnTurnExtCycle`:

```
public override void OnTurnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

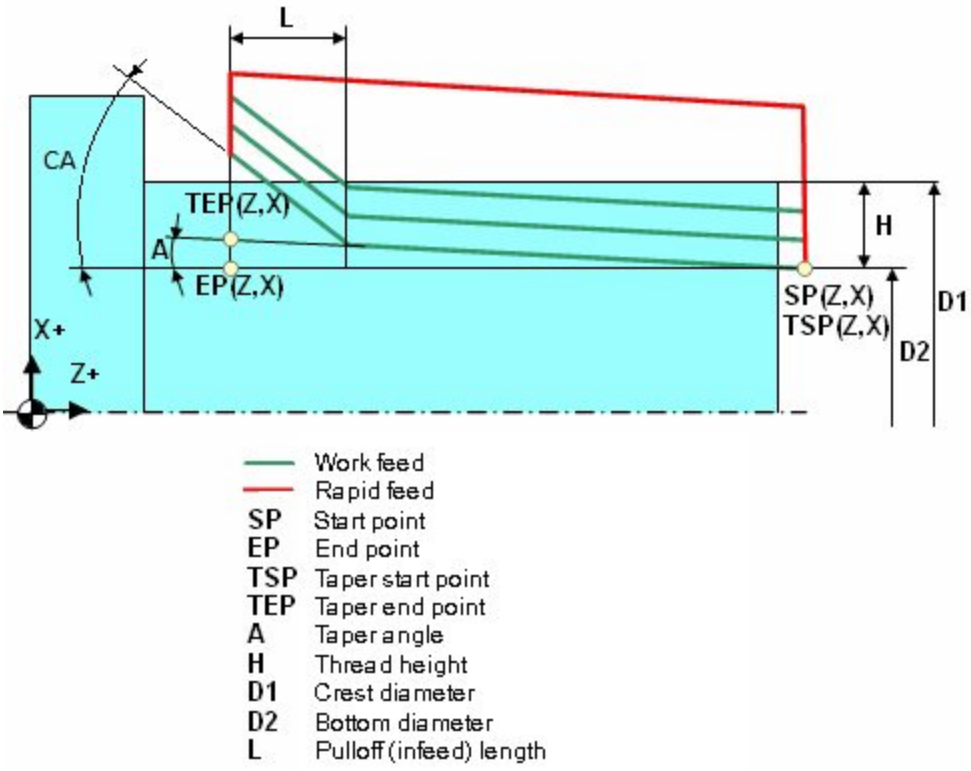
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE LATHETHREAD - Lathe threading cycle (G76)

`LATHETHREAD(403)` Lathe multipass threading canned cycle G76 is used to form a single line command that specifies all threading parameters. The desired threading depth is achieved by automatic multiple passes. Cycle parameters include threading start and end point coordinates, conical angle (for taper threading), chamfer dimensions, pressure angles, threading depth, pass count, cut-in strategy, etc.



Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: LATHETHREAD(403).
CLD[3]	CLD.CLParams(1)	The thread direction: 0 – left, 1 – right.

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	The thread orientation: 0 – outer diameter, 1 – inner diameter, 2 – face.
CLD[5]	CLD.CLParams(3)	The cone angle in degrees (for a taper thread) (A).
CLD[6]	CLD.CLParams(4)	The Z coordinate of the thread start point (SP.Z).
CLD[7]	CLD.CLParams(5)	The X coordinate of the thread start point (SP.X).
CLD[8]	CLD.CLParams(6)	The Z coordinate of the thread end point (EP.Z).
CLD[9]	CLD.CLParams(7)	The X coordinate of the thread end point (EP.X).
CLD[10]	CLD.CLParams(8)	The Z coordinate of the taper thread start point (TSP.Z).
CLD[11]	CLD.CLParams(9)	The X coordinate of the taper thread start point (TSP.X).
CLD[12]	CLD.CLParams(10)	The Z coordinate of the taper thread end point (TEP.Z).
CLD[13]	CLD.CLParams(11)	The X coordinate of the taper thread end point (TEP.X).
CLD[14]	CLD.CLParams(12)	The pulloff length at the thread termination (L).
CLD[15]	CLD.CLParams(13)	The thread type: 0 – special, 1 – metric, 2 – cylinder pipe thread, 3 – trapezium thread, 4 – stop thread, 5 – round thread, 6 – cone pipe thread, 7 – internal cone thread, 8 – internal

CLD array	CLD name	Description
		cylinder thread, 9 – cone imperial thread.
CLD[16]	CLD.CLParams(14)	The crest diameter (D1).
CLD[17]	CLD.CLParams(15)	The bottom diameter (D2).
CLD[18]	CLD.CLParams(16)	The thread height (H).
CLD[19]	CLD.CLParams(17)	The thread profile angle between two neighboring thread edges
CLD[20]	CLD.CLParams(18)	The thread profile angle between a vertical line and a thread edge
CLD[21]	CLD.CLParams(19)	The number of the thread starts.
CLD[22]	CLD.CLParams(20)	The thread step definition mode: 0 – by distance, 1 – by number of the threads per unit.
CLD[23]	CLD.CLParams(21)	The thread step amount.

CLD array	CLD name	Description
CLD[24]	CLD.CLParams(22)	The thread sequence: 0 – radial 1 – Flank 2 – Alternate flank
CLD[25]	CLD.CLParams(23)	The cut depth calculation method: 0 – keep the cutting area constant 1 – keep the cutting depth constant
CLD[26]	CLD.CLParams(24)	The cut depth definition mode: 0 – by the cut depth of the first cut, 1 – by the number of cuts.
CLD[27]	CLD.CLParams(25)	The first cut depth (is used together with the "0" cut depth definition method).
CLD[28]	CLD.CLParams(26)	The number of depth cuts (is used together with the "1" cut depth definition mode).

CLD array	CLD name	Description
CLD[29]	CLD.CLParams(27)	The finish cut depth.
CLD[30]	CLD.CLParams(28)	The Spring cut number.
CLD[31]	CLD.CLParams(29)	The rough pass minimal depth
CLD[32]	CLD.CLParams(30)	The pulloff angle at the thread termination in degrees (CA).
CLD[33]	CLD.CLParams(31)	The spindle start angle in degrees (for multistart threads).
CLD[34]	CLD.CLParams(32)	Thread engage length along turn axis (like L size on the picture but at the start of thread).
CLD[35]	CLD.CLParams(33)	Tool plunge angle in degrees which depends on CLD[24] also.
CLD[36]	CLD.CLParams(34)	Thread retract length along turn axis (like L size on the picture but it does not depend on retract angle)

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the turning-family handler `OnTurnExtCycle`:

```
public override void OnTurnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE LATHETHREADG92(404) - Lathe threading single pass cycle (G92)

Lathe threading single pass cycle G92 `LATHETHREADG92(404)` (in different controls may be termed G92, G78, G21, etc.) generates a closed sequence of moves for a single pass threading. Illustration of execution is shown below. Before calling the cycle tool is located at the Start point. The cycle is called one frame of the NC program, which specifies the endpoint, thread pitch, taper and chamfer size, etc. As a result of this frame tool from the Start point will fall to the TSP point, make threading to the TEP point and come to the Start point. Because thread cutting is usually performed in several passes, the NC program often consists of multiple calls to the cycle with different values of the diameter of the thread:

```
...
X60.0 Z20.0 M08
G01 Z10.0 F1.0 (Approach to the Start point)
G92 X29.4 Z-52.0 F2.0 (Cycle call)
X28.9 (Modal call G92 cycle with a different value of diameter)
X28.5 (Modal call G92 cycle with a different value of diameter)
X28.1 (Modal call G92 cycle with a different value of diameter)
X27.8 (Modal call G92 cycle with a different value of diameter)
X27.56 (Modal call G92 cycle with a different value of diameter)
X27.36 (Modal call G92 cycle with a different value of diameter)
X27.26 (Modal call G92 cycle with a different value of diameter)
G00 X200.0Z150.0M09 (Return of the tool)
...
```

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: LATHETHREADG92(404).
CLD[4]	CLD.CLParams(2)	The thread orientation: 0 – outer diameter, 1 – inner diameter, 2 – face.

CLD array	CLD name	Description
CLD[5]	CLD.CLParams(3)	The cone angle in degrees (for a taper thread) (A).
CLD[6]	CLD.CLParams(4)	The Z coordinate of the thread start point (SP.Z).
CLD[7]	CLD.CLParams(5)	The X coordinate of the thread start point (SP.X).
CLD[8]	CLD.CLParams(6)	The Z coordinate of the thread end point (EP.Z).
CLD[9]	CLD.CLParams(7)	The X coordinate of the thread end point (EP.X).
CLD[10]	CLD.CLParams(8)	The Z coordinate of the taper thread start point (TSP.Z).
CLD[11]	CLD.CLParams(9)	The X coordinate of the taper thread start point (TSP.X).
CLD[12]	CLD.CLParams(10)	The Z coordinate of the taper thread end point (TEP.Z).
CLD[13]	CLD.CLParams(11)	The X coordinate of the taper thread end point (TEP.X).
CLD[14]	CLD.CLParams(12)	The pulloff length at the thread termination (L).
CLD[21]	CLD.CLParams(19)	The number of the thread starts.
CLD[22]	CLD.CLParams(20)	The thread step definition mode: 0 – by distance, 1 – by number of the threads per unit.
CLD[23]	CLD.CLParams(21)	The thread step amount.
CLD[32]	CLD.CLParams(30)	The pulloff angle at the thread termination in degrees (CA).

CLD array	CLD name	Description
CLD[33]	CLD.CLParams(31)	The spindle start angle in degrees (for multistart threads).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the turning-family handler `OnTurnExtCycle`:

```
public override void OnTurnExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

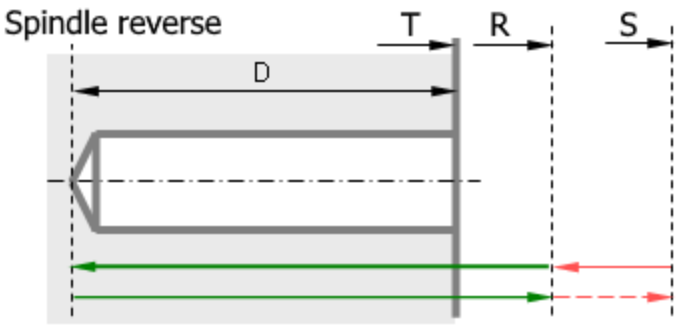
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE TAP - Lathe tapping cycle (G84)

Lathe tapping cycle `TAP(168)` G84 is used to machine threading by static tap in an axial hole of the workpiece which is rotated by the spindle. The tool approaches the hole axis at the safe plane level, rapid travels to the return level, then work feedrate equal to the threading pitch machines the threading to the specified depth. At the hole bottom the spindle rotation is reversed and the tool is retracted at work feedrate to the return level, where the spindle rotation direction and frequency are reset.



Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: TAP(168).
CLD[3]	CLD.CLParams(1)	Upper level of hole.
CLD[4]	CLD.CLParams(2)	Hole depth from upper level.
CLD[5]	CLD.CLParams(3)	Return level.
CLD[6]	CLD.CLParams(4)	Safe level.
CLD[7]	CLD.CLParams(5)	Reserved.
CLD[8]	CLD.CLParams(6)	Reserved.
CLD[9]	CLD.CLParams(7)	Work feed (thread pitch)
CLD[10]	CLD.CLParams(8)	Return feed.

Access from .NET

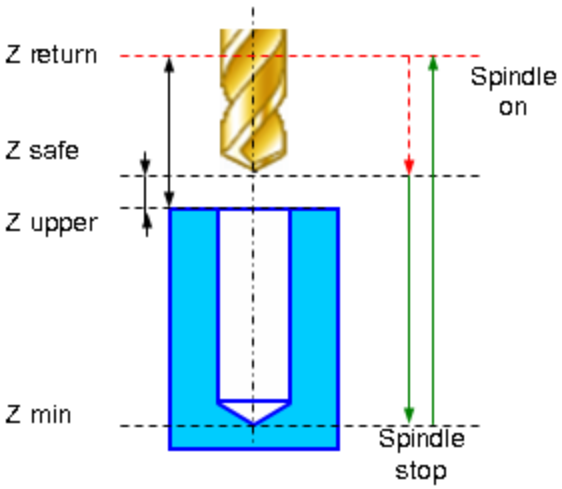
This cycle has no dedicated family handler - it is delivered through the base `OnExtCycle` (dispatch on `cmd.CycleType` / `cld[2]`). See Extended cycle (EXTCYCLE).

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DBore5 - Drilling cycle (G85)

Boring cycle `W5DBore5(485)` (G85) performs tool approach to the hole center, hole boring with stop at minimum level and work feedrate retract to the Return level.



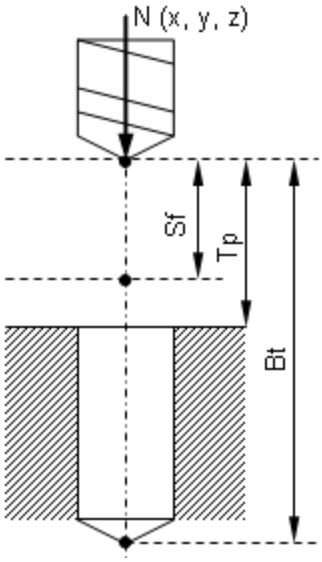
Boring canned cycle G85 consists of:

- Rapid travel to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate travel to the Z min level.
- Spindle stop.
- Work feedrate return to the Z return level.
- Restore spindle rotation direction and speed.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DBore5(485).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).



Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

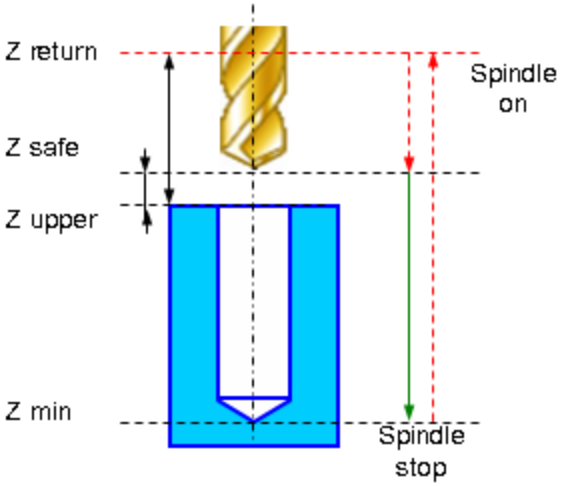
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DBore6 - Drilling cycle (G86)

Boring cycle `W5DBore6(486)` (G86) performs tool approach to the hole center, hole boring with stop at minimum level and rapid retract to the Return level.



Boring canned cycle G86 consists of:

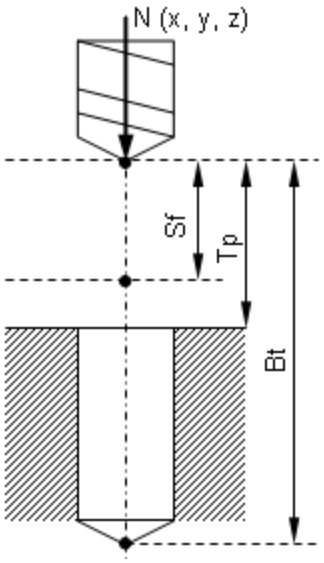
- Rapid travel to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate travel to the Z min level.
- Spindle stop. If oriented spindle stop is enabled (CLD[17] parameter), it is positioning the spindle in a given angular position and then tool shifts to specified small value of the X, Y and Z coordinates.
- Rapid return to the Z return level.
- Restore spindle rotation direction and speed.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DBore6(486) (G86).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[16]	CLD.CLParams(14)	Delay at the top level in seconds
CLD[17]	CLD.CLParams(15)	The state of the oriented spindle stop at the bottom of the hole: 0 - off, 1 - enabled.

CLD array	CLD name	Description
CLD[18]	CLD.CLParams(16)	The angular position of the spindle when stopping at the bottom of the hole (degrees)
CLD[19]	CLD.CLParams(17)	The value of tool shift for the X coordinate after oriented spindle stop at the bottom of the hole.
CLD[20]	CLD.CLParams(18)	The value of tool shift for the Y coordinate after oriented spindle stop at the bottom of the hole.
CLD[21]	CLD.CLParams(19)	The value of tool shift for the Z coordinate after oriented spindle stop at the bottom of the hole.
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).



Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

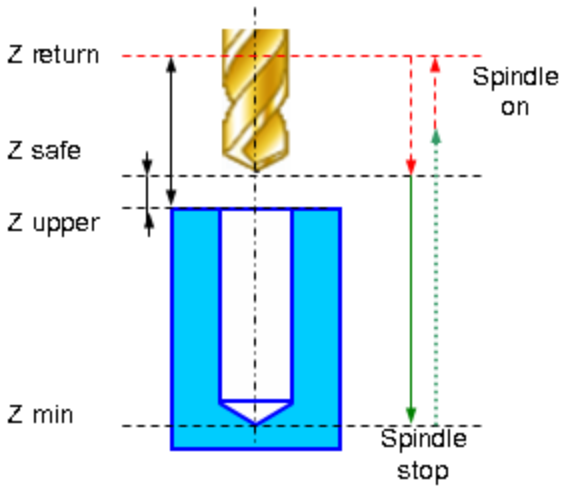
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DBore7 - Drilling cycle (G87)

Boring cycle `W5DBore7(487)` (G87) performs tool approach to the hole center, hole boring with stop at minimum level and manual retract to the Return level.



Boring canned cycle G87 consists of:

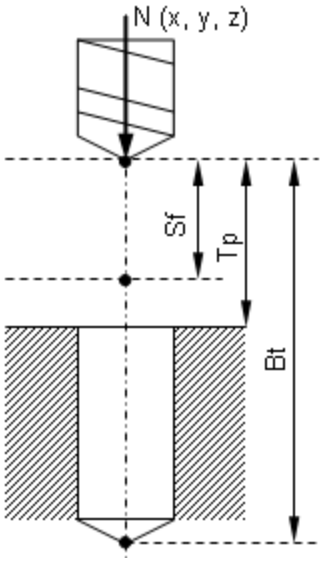
- Rapid travel to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate travel to the Z min level.
- Spindle stop. If oriented spindle stop is enabled (CLD[17] parameter), it is positioning the spindle in a given angular position and then tool shifts to specified small value of the X, Y and Z coordinates.
- Manual retract to the Z return level.
- Restore spindle rotation direction and speed.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DBore7(487) (G87).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[17]	CLD.CLParams(15)	The state of the oriented spindle stop at the bottom of the hole: 0 - off, 1 - enabled.
CLD[18]	CLD.CLParams(16)	The angular position of the spindle when stopping at the bottom of the hole (degrees)

CLD array	CLD name	Description
CLD[19]	CLD.CLParams(17)	The value of tool shift for the X coordinate after oriented spindle stop at the bottom of the hole.
CLD[20]	CLD.CLParams(18)	The value of tool shift for the Y coordinate after oriented spindle stop at the bottom of the hole.
CLD[21]	CLD.CLParams(19)	The value of tool shift for the Z coordinate after oriented spindle stop at the bottom of the hole.
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).



Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

See Extended cycle (EXTCYCLE) for the common .NET model.

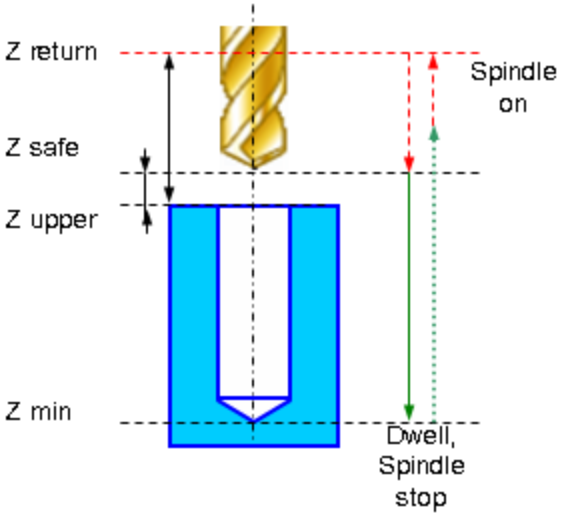
See also

- Extended cycle (EXTCYCLE)

- CLData access model

EXTCYCLE W5DBore8 - Drilling cycle (G88)

Drilling cycle type `W5DBore8(488)` (G88) drills holes with rapid approach to the safe level, dwell at hole bottom level, spindle stop and manual retract to the safe plane level.



Drilling cycle G88 consist of the following steps:

- Rapid tool motion to the hole center at the Z return level.
- Rapid descend to the Z safe level.
- Work feedrate motion to the Z min level.
- Dwell at the Z min level.
- Spindle stop.
- Manual tool retract to the Z return level.
- Restore the spindle rotation direction and speed.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DBore8(488) (G88).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

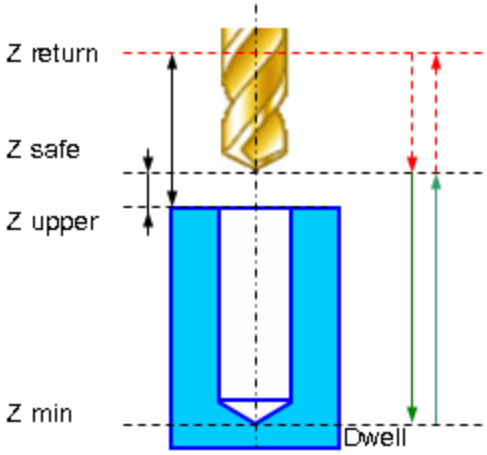
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DBore9 - Drilling cycle (G89)

Boring cycle type `W5DBore9(489)` (G89) Bores holes with rapid approach to the safe level, dwell at hole bottom level, spindle stop and manual retract to the safe plane level.



Boring cycle G89 consist of the following steps:

- Rapid tool motion to the hole center at the Z return level.
- Rapid descend to the Z safe level.
- Work feedrate motion to the Z min level.
- Dwell at the Z min level.
- Work feedrate return to the Z safe level.
- Restore the spindle rotation direction and speed.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DBore9(489) (G89).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

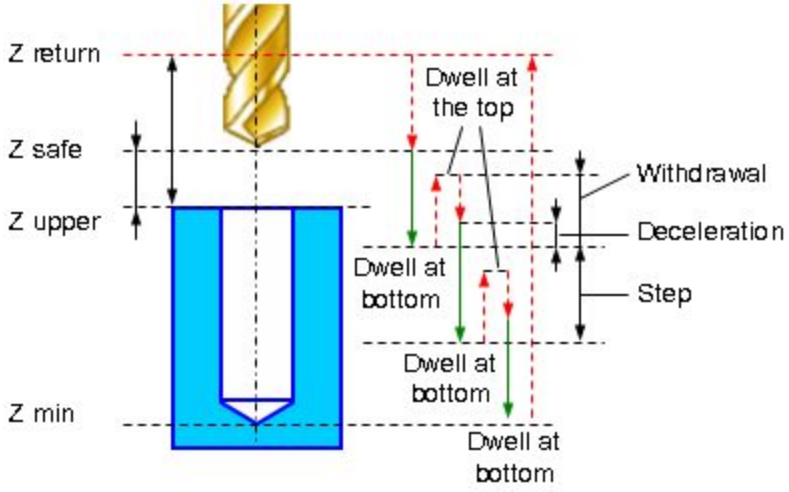
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DChipBreaking - Drilling with chip breaking cycle (G73)

Drilling with chip breaking cycle `W5DChipBreaking(473)` (G73) performs tool approach to the hole center at the Z return level. When cyclic drilling is performed with tool retraction for chip breaking.



The cycle consists of:

- Rapid approach to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate motion to the Step depth S.
- Dwell for the Delay at the bottom time.
- Rapid tool retraction for the `Withdrawal distance (Ld)`.
- Dwell for the Dwell at the top time.
- Rapid motion to the previous depth level, with a `Deceleration (Dcl)`.
- Work feedrate to the `Deceleration (Dcl)` with `Step (S)`.
- Dwell at for the Dwell at the bottom time.
- Repeat previous five iterations until the full hole depth is reached.
- Rapid tool return to the Z retract level..

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DChipBreaking(473).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min

CLD array	CLD name	Description
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[16]	CLD.CLParams(14)	Delay at the top level in seconds
CLD[17]	CLD.CLParams(15)	St, chip breaking step value
CLD[18]	CLD.CLParams(16)	Dg, each iteration chip breaking step degression
CLD[19]	CLD.CLParams(17)	Dc, before drilling deceleration value for each step
CLD[20]	CLD.CLParams(18)	Ld, each step lead out value for cutting process breaking
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

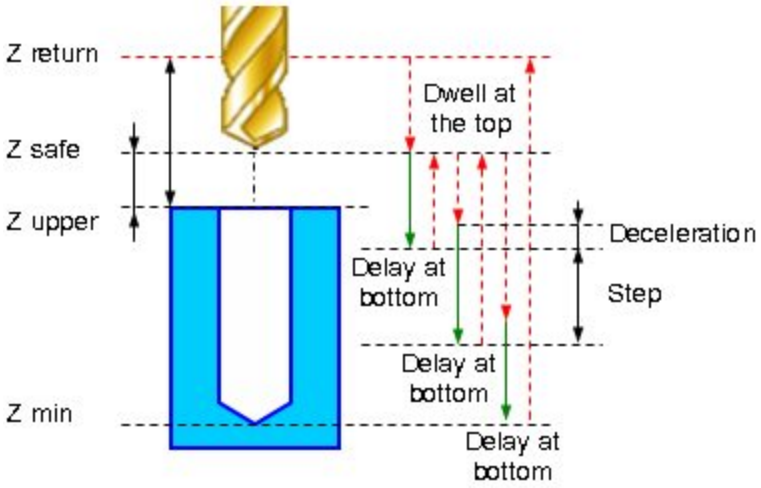
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DChipRemoving - Drilling with chip removing cycle (G83)

Drilling with chip removing cycle `W5DChipRemoving(483)` (G83) performs tool motion to the hole center at the Z return level and consequent cyclic drill with tool retraction to the Z safe level.



Cycle consists of:

- Rapid approach to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate motion to the Step depth S.
- Dwell for the Delay at the bottom time.
- Rapid return to Z safe level.
- Dwell for the Dwell at the top time.
- Rapid motion to the previous depth level, with a `Deceleration (Dcl)`.
- Work feedrate to the `Deceleration (Dcl)` with `Step (S)`.
- Dwell at for the Dwell at the bottom time.
- Repeat previous five iterations until the full hole depth is reached.
- Rapid tool return to the Z retract level.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.

CLD array	CLD name	Description
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DChipRemoving(483).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds

CLD array	CLD name	Description
CLD[16]	CLD.CLParams(14)	Delay at the top level in seconds
CLD[17]	CLD.CLParams(15)	St, chip removing step value
CLD[18]	CLD.CLParams(16)	Dg, each iteration chip removing step degression
CLD[19]	CLD.CLParams(17)	Dc, before drilling deceleration value for each step
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

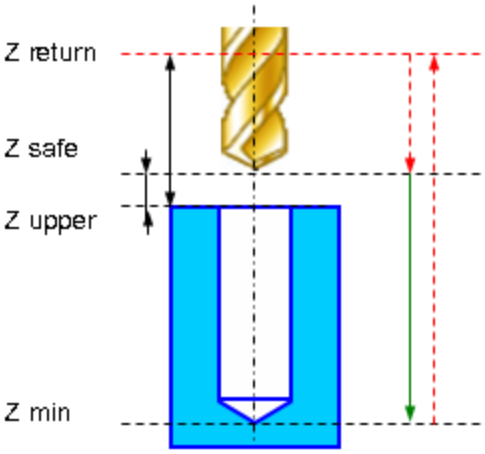
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DDrill - Drilling cycle (G81)

Drilling cycle type `W5DDrill(481)` (G81) drills holes with rapid approach to the safe level and rapid retract the safe plane level.



Drilling cycle G81 consist of the following steps:

- Rapid tool motion to the hole center at the Z return level.
- Rapid descend to the Z safe level.
- Work feedrate motion to the Z min level.
- Rapid tool return to the Z return level.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DDrill(481).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

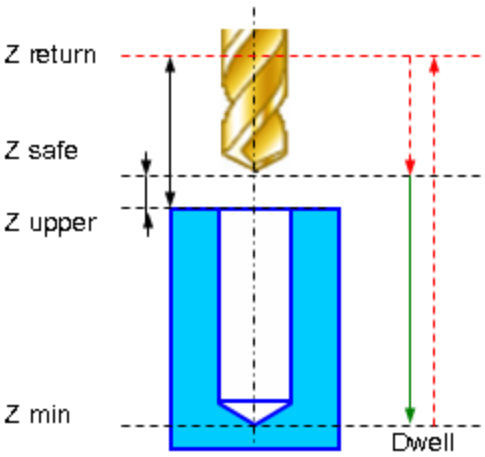
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DFace - Drilling cycle (G82)

Drilling cycle type `W5DFace(482)` (G82) drills holes with rapid approach to the safe level, dwell at hole bottom level and rapid retract the safe plane level.



Drilling cycle G82 consists of the following steps:

- Rapid tool motion to the hole center at the Z return level.
- Rapid descend to the Z safe level.
- Work feedrate motion to the Z min level.
- Dwell at the Z min level.
- Rapid tool return to the Z return level.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DFace(482).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

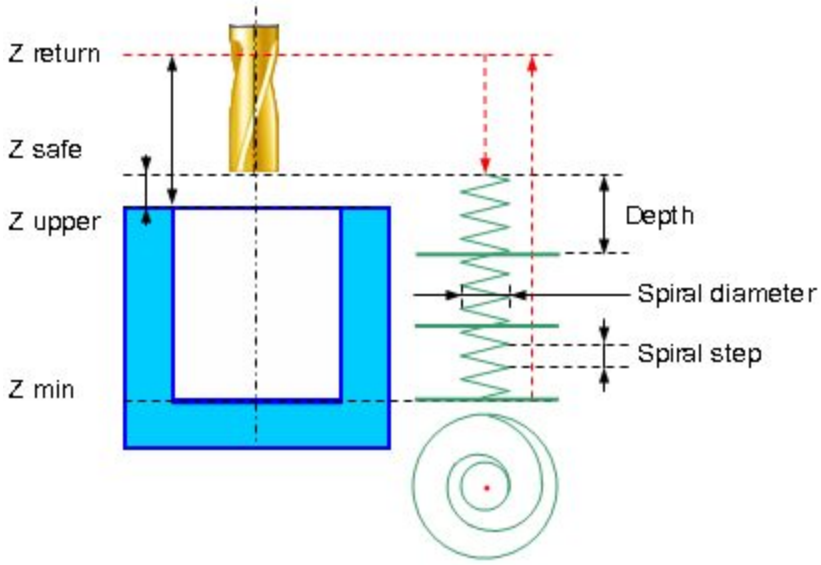
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DHolePocketing - Hole pocketing cycle

`W5DHolePocketing(491)` is used to machine holes whose diameter is greater than the tool diameter. The pocketing is performed by layers. The tool cuts in along a spiral to each layer and then expands the hole to the desired diameter by moving along Archimedes spiral with finishing pass along the circle.



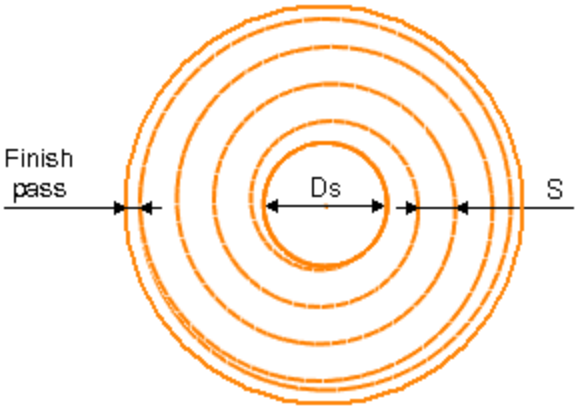
Hole pocketing cycle includes the following:

- Rapid approach to the center of the hole at the Z retract level.
- Rapid motion to the Z safe level.
- Work feedrate spiral cut-in to the Z machining depth.
- Archimedes spiral with Step S motion at that level until the tool axis has reaches the circle with diameter equal to hole diameter reduced by the tool diameter.
- Finishing pass along specified above circle without level change.
- Repeat previous three steps until desired hole depth is machined with travel to the next cut-in point without level change.
- Return to the hole center.
- Rapid travel to the Z retract level.

Spiral twist direction can be one of the following:

- Right. The Spiral is twisted right. The tool is rotating clockwise if watched from above.
- Left. The spiral is twisted left. The tool is rotating counter clockwise if watched from above..

- Counter. Spiral twist direction is determined by the spindle rotation direction and corresponds to the up cutting milling. When counter pocketing milling tool rotation direction and spiral direction are opposite to each other.
- Follow. Spiral twist direction is determined by the spindle rotation direction and corresponds to the down cutting milling. When counter pocketing milling tool rotation direction and spiral direction are coincident.



If the finishing pass value is not zero, then additional pass along the circle with specified stock is performed before the final pass along the circle. This allows to ensure equal stock final pass.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – cycle on, CALL(52) – cycle call, OFF(72) – cycle off.
CLD[2]	CLD.SubType	Cycle type identifier: W5DHolePocketing(491).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level

CLD array	CLD name	Description
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[16]	CLD.CLParams(14)	Hole diameter
CLD[17]	CLD.CLParams(15)	Spiral step
CLD[18]	CLD.CLParams(16)	Spiral diameter
CLD[19]	CLD.CLParams(17)	Spiral direction: 0 – counter, 1 – follow, 2 – right, 3 – left. In the 0 and 1 cases spiral direction depend on the tool rotation direction, in the 2 and 3 cases they are independent.
CLD[20]	CLD.CLParams(18)	Cutting depth along tool axis
CLD[21]	CLD.CLParams(19)	Axial cut count
CLD[22]	CLD.CLParams(20)	Archimedean spiral step

CLD array	CLD name	Description
CLD[23]	CLD.CLParams(21)	Finish pass value

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

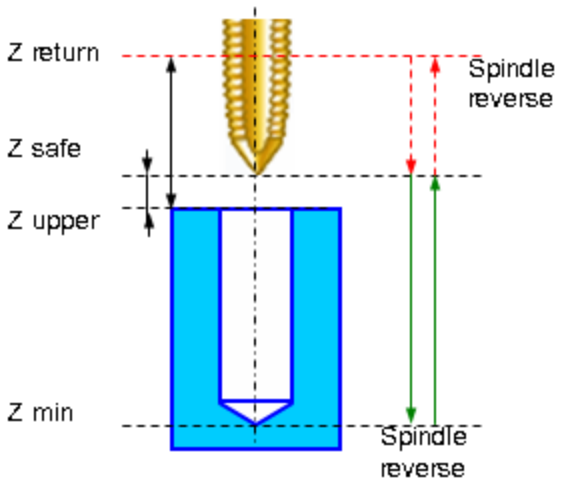
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DTap - Tapping cycle (G84)

Tapping cycle `W5DTap(484)` (G84) performs rapid approach to the Z return level, thread tapping with subsequent retraction at work feedrate with reverse spindle rotation.



G84 tapping cycle includes:

- Rapid approach to the hole center at the Z return level.
- Rapid travel to the Z safe level.
- Work feedrate motion the Z min level.
- Spindle reverse rotation and work feedrate trave to Z safe level.
- Rapid retract to the Z return level.
- Restore spindle rotation direction and speed.

If the appropriate setting is enabled, then tap retracts to break / remove chip during the cycle execution. Way to break chips is determined by CLD [20] parameter.

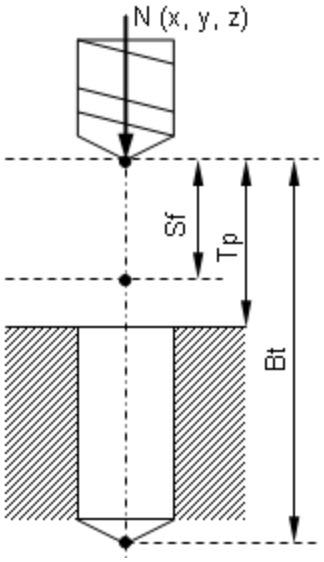
CNC-systems often have different tapping canned cycles for fixed and floating socket types. Socket type is specified by the `CLD[19]` (CLD.CLParams(17)) parameter.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – canned cycle on, CALL(52) – canned cycle call, OFF(72) – canned cycle off.
CLD[2]	CLD.SubType	Canned cycle type identifier: W5DTap(484).

CLD array	CLD name	Description
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[15]	CLD.CLParams(13)	Delay at the bottom level in seconds
CLD[16]	CLD.CLParams(14)	Delay at the top level in seconds
CLD[17]	CLD.CLParams(15)	Thread step

CLD array	CLD name	Description
CLD[18]	CLD.CLParams(16)	Initial spindle angle in degrees (for multistart threads)
CLD[19]	CLD.CLParams(17)	Tap socket type: 0 – floating (compensative), 1 – fixed (uncompensative)
CLD[20]	CLD.CLParams(18)	The way to break chip: 0 – without chip breaking, 1 – chip removing, 2 – chip breaking
CLD[21]	CLD.CLParams(19)	St, chip breaking step value
CLD[22]	CLD.CLParams(20)	Dg, each iteration chip breaking step degression
CLD[23]	CLD.CLParams(21)	Dc, before drilling deceleration value for each step
CLD[24]	CLD.CLParams(22)	Ld, each step lead out value for cutting process breaking
CLD[50]	CLD.CLParams(48)	What spindle is used to machining: 1 - driven tool, 2 - workpiece spindle (lathe).



Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

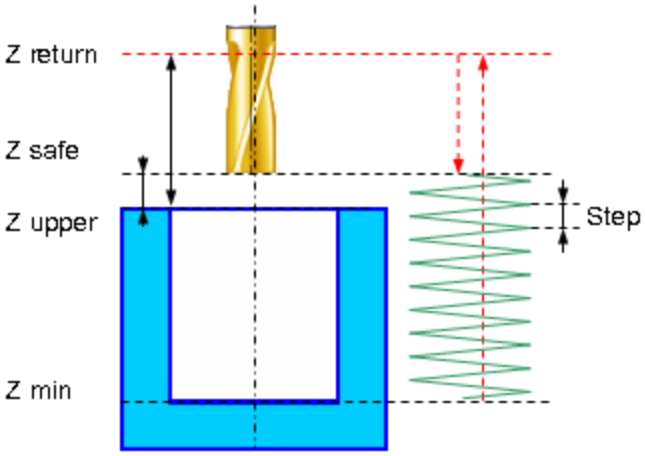
See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE W5DThreadMill - Thread milling cycle

Thread milling `W5DThreadMill(490)` cycle is used to machine external or internal threading or to machine hole by a helix. Spiral machining is used when hole diameter is larger than the tool diameter. The tool rotates around the hole axis and simultaneously travels along the axis. spiral diameter is chosen according to the hole and the tool dimensions. Machining can be done in several passes to mill holes of desired diameter.



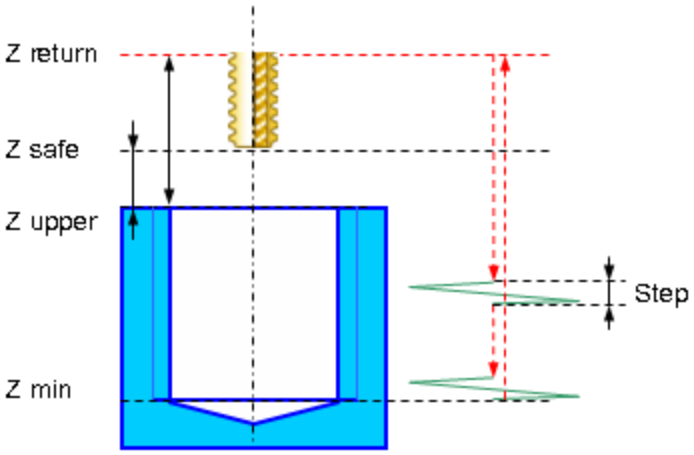
Spiral machining includes the following steps:

- Rapid approach to the hole center at the Z retract level.
- Rapid travel to the Z safe level.
- Work feedrate travel to the spiral start.
- Work feedrate spiral motion to the Z min level.
- Optional circular pass on the bottom level. Circle diameter equals spiral diameter.
- Return to the hole center.
- Rapid travel to the Z safe level.
- If additional roughing and finishing passes are applied previous five steps are repeated until desired hole diameter is reached.
- Rapid travel to the Z retract level.

Threadmilling provides the following advantages over traditional tapping:

- blind and through, left and right threads are machined by the same tool;
- different threads with the same pitch are machined by the same tool;
- all precision parameters are secured by the same tool;
- accurate threading is machined to the full depth of the blind hole as the mill has no chamfer;
- different materials are machined by the same tool;
- high reliability of the machining because of good chip handling;
- high efficiency of threadmilling due to higher cutting speed and feedrate;
- low spindle torque even for coarse thread machining.

For threadmilling both single-cutter tools and multi-cutter ones allowing to machine several thread turns in one pass. Multi-cutter tool machining is much similar to spiral machining.



When using the multi-cutter tool threadmilling machining includes the following steps:

- Rapid approach to the hole center at the Z retract level.
- Rapid travel to the Z safe level.
- Rapid travel to the tool cutting edge length distance which is determined by the number and size of the mill tooth size (thread pitch).
- Work feedrate travel to the start of the spiral.
- Machining along one spiral turn with step equaling thread pitch.
- Retract to the hole center.
- If one spiral turn is not enough to machine the threading to the full hole depth descend to the cutting-edge length and spiral motion are repeated until desired threading depth is reached.
- Rapid return to the Z retract level.

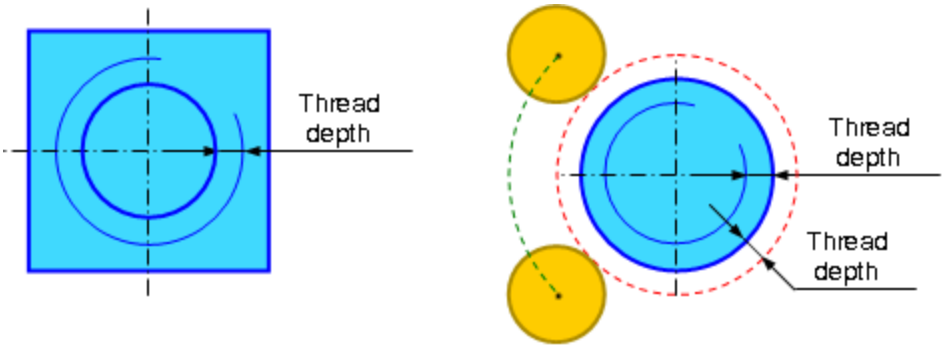
If additional roughing and finishing passes are applied then the above steps are repeated until specified thread depth is achieved.

Thread type parameter specifies whether the threading is External or Internal.

There are cases when technology requires that threading is machined upside down and there are cases when threading is done from the bottom to the top. This cycle supports both technologies.

Thread kind, right or left, is determined by the Thread spiral direction parameter. For spiral machining it is convenient to define the spiral direction according to the spindle rotation direction. When Follow direction is specified the tool rotation and the spiral directions coincide, for the Counter direction they are opposite.

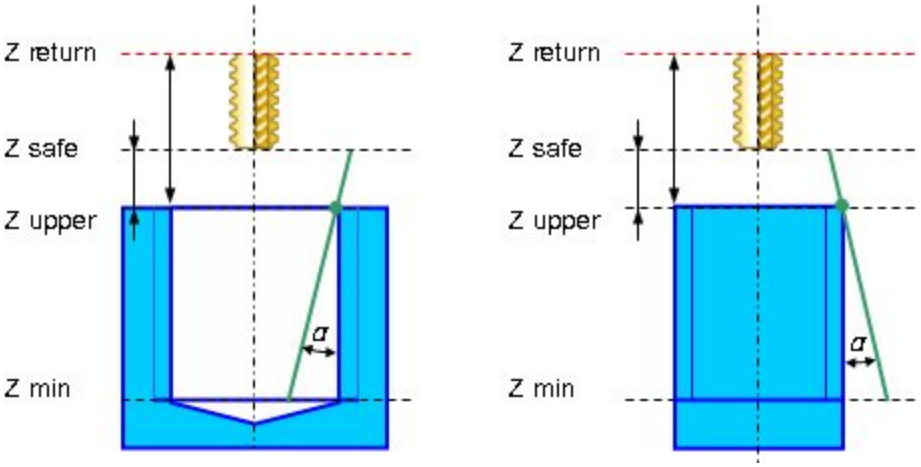
Threading depth parameter, is used to specify the difference between outer and inner diameters of the thread.



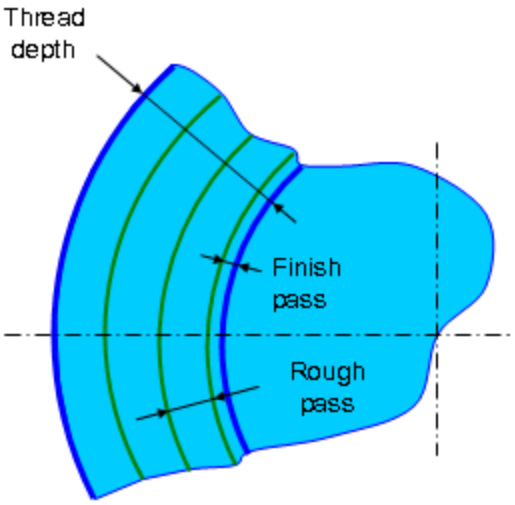
Bottom circle pass parameter specifies whether the circle motion is performed when the bottom of the thread is reached.

Multi-start thread is machined if Thread start count parameter is greater than 1. If start count is 1 single-start thread is machined.

Thread milling cycle supports tapered thread machining. Taper angle parameter is used to specify the thread taper angle in degrees.The taper angle is measured from the top level of the hole (lug). Positive angle direction for tapered thread machining of the hole is the direction to the center of the hole. Positive angle direction for tapered thread machining of the lug is the direction from the center of the lug.



Threadmilling can be performed by several passes. Use the Roughing passes count, Roughing pass step and Finish pass step parameters to process this machining cycles.



Toolpath type parameter is used defines the toolpath type according to the used tool type. It can be one of the following

- Continuous. This toolpath type is used for the single-cutter tool, which forms only one thread turn with each turn of the spiral. Geometrically the toolpath is a continuous spiral.
- Transition along the axis. This type is used for multi-cutter tool which forms multiple thread turns for one turn of the toolpath spiral. The toolpath consists of subsequent spiral turns connected with rapid cutting-edge long transitions along the spiral axis.

Cutting edge length is used only for Transition along the axis toolpath type. It specifies the length of transitions between adjacent spiral turns.

The Number of spiral turns parameter is used to machine threads with pitch smaller than the distance between the tool's teeth. This parameter is generally unused (equals 1).

The Start angular position parameter specifies the initial angular position of the spindle in degrees at which thread milling starts.

Parameters

CLD array	CLD name	Description
CLD[1]	CLD.SubCmd	Command type: ON(71) – cycle on, CALL(52) – cycle call, OFF(72) – cycle off.
CLD[2]	CLD.SubType	Cycle type identifier: W5DThreadMill(490).
CLD[3]	CLD.CLParams(1)	Nx, X coordinate of the tool normal vector

CLD array	CLD name	Description
CLD[4]	CLD.CLParams(2)	Ny, Y coordinate of the tool normal vector
CLD[5]	CLD.CLParams(3)	Nz, Z coordinate of the tool normal vector
CLD[6]	CLD.CLParams(4)	Sf, Normal distance from current tool position to the safe plane level
CLD[7]	CLD.CLParams(5)	Tp, Normal distance from current tool position to the hole top level
CLD[8]	CLD.CLParams(6)	Bt, Normal distance from current tool position to the hole bottom level
CLD[9]	CLD.CLParams(7)	Work feed measurements: 0 – mm/rev, 1 – mm/min
CLD[10]	CLD.CLParams(8)	Work feed value
CLD[11]	CLD.CLParams(9)	Approach feed measurements: 0 – mm/rev, 1 – mm/min
CLD[12]	CLD.CLParams(10)	Approach feed value
CLD[13]	CLD.CLParams(11)	Return feed measurements: 0 – mm/rev, 1 – mm/min
CLD[14]	CLD.CLParams(12)	Return feed value
CLD[16]	CLD.CLParams(14)	D, thread outer diameter
CLD[17]	CLD.CLParams(15)	St, thread step
CLD[18]	CLD.CLParams(16)	Thread type: 0 – inner, 1 – outer
CLD[19]	CLD.CLParams(17)	Thread spiral direction: 0 – counter, 1 – follow, 2 – right, 3 – left. In the 0 and 1 cases

CLD array	CLD name	Description
		thread direction depend on the tool rotation direction, in the 2 and 3 cases they are independent.
CLD[20]	CLD.CLParams(18)	Machining sequence: 0 – top-down, 1 – bottom-up
CLD[21]	CLD.CLParams(19)	Thread start count
CLD[22]	CLD.CLParams(20)	Thread depth (difference between outer and inner radiuses)
CLD[23]	CLD.CLParams(21)	Rough passes step
CLD[24]	CLD.CLParams(22)	Rough passes count
CLD[25]	CLD.CLParams(23)	Finish pass step
CLD[26]	CLD.CLParams(24)	Bottom circle pass necessity: 0 – do not make last circle pass, 1 – make last circle pass
CLD[27]	CLD.CLParams(25)	Taper angle in degrees
CLD[28]	CLD.CLParams(26)	Method of forming a spiral: 0 – continuously through the entire depth of the holes, 1 – with transition for multitoothed tool
CLD[29]	CLD.CLParams(27)	The length of the working part of cutters (determined by the number of teeth). This value determines the length of the transition to the method of forming a spiral "with transition".
CLD[30]	CLD.CLParams(28)	The number of revolutions of the spiral to the method of forming a spiral "with transition" (usually

CLD array	CLD name	Description
		equal to 1, but can be greater than 1 when milling the thread with step smaller than cutter step).
CLD[31]	CLD.CLParams(29)	The initial angular position of the spindle in degrees

Access from .NET

In addition to the base `OnExtCycle`, this cycle is delivered through the hole-family handler `OnHoleExtCycle`:

```
public override void OnHoleExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

See Extended cycle (EXTCYCLE) for the common .NET model.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

EXTCYCLE WProbing - Probing cycle

`WProbing` is used for probing operations. It measures a part or tool based on one of the selected cycles using approach and return to the beginning of the cycle. Each cycle has its own toolpath, touch points and parameters on the basis of which further calculations are made.

`WProbing` stores parameters by code, and not by index, like other cycles of the `EXTCYCLE` command.

Below are examples of syntax for working:

Postprocessor (sppx):

```
program ExtCycle
  if cld[1]=52 then begin ! Cycle call
    case cld[2] of
      500: begin ! All probing cycles
        case cmdPrm.Int[-2] of ! Probing cycle sub type
          11: begin ! My hole cycle P9814
            Output "G65 P9814 E" + cmdPrm.Flt[101] + " Q" + cmdPrm.Flt[102]
          end
          12: begin ! My boss cycle P9812
            Output "G65 P9812 E" + cmdPrm.Flt[101] + " Q" + cmdPrm.Flt[102]
          end
        end
      end
    end ! End of All probing cycles
  end
end
```

.NET postprocessor:

```
public override void OnProbeExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
{
    if (!cmd.IsCall)
        return;
    switch (cmd.Prm.Int[-2]) // Cycle type
    {
        case 11: //My hole cycle P9814
            nc.WriteLine($"G65 P9814 E{cmd.Prm.Flt[101]} Q{cmd.Prm.Flt[102]}");
            break;
        case 12: //My boss cycle P9812
            nc.WriteLine($"G65 P9812 E{cmd.Prm.Flt[101]} Q{cmd.Prm.Flt[102]}");
            break;
```

```
    }
}
```

You can see a more detailed example of measurement cycles inside the distribution Fanuc (30i)_Mill_DN.dll postprocessor.

Subtype of probing cycles has "500" value.

cmd operator (parameters with a unique code) is described here.

OnProbeExtCycle handler

Separate `OnProbeExtCycle` handler has been added for.net postprocessor which performs on EXTCYCLE CLData command for the probing cycles case (which SubCode is 500).

Declaration of `OnProbeExtCycle` handler:

```
public virtual void OnProbeExtCycle(ICLDExtCycleCommand cmd, CLDArray cld)
```

Parameters

Type	Name	Description
ICLDExtCycleCommand	cmd	The current CLData command.
CLDArray	cld	The cld array of the command.

More information about probing operations and cycles you can find in user manual.

See also

- Extended cycle (EXTCYCLE)
- CLData access model

Additional parameters

NCAction

`NCAction` element has one own parameters:

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: NCAction value = 1
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab

Custom properties

Custom properties have their own parameter codes that are stored in cycles and `NCAction` element:

CLDArray	Type	Description
CmdPrm.Int[-10]	Integer	Set WCS offset - mode. 0 - Global (all G54-G59); 1 - One WCS (one of G54-G59); 2 - Parametrical offset (G10 L2 P1 XYZ); 3 - Local CS offset (G52 XYZ)
CmdPrm.Int[-11]	Integer	Set WCS offset - CS Number (Set WCS offset - mode = 1 or 2).
CmdPrm.Int[-12]	Integer	Set offset of tool number - Tool number
CmdPrm.Int[-13]	Integer	Set offset of tool number - Corrector 1
CmdPrm.Int[-14]	Integer	Set offset of tool number - Corrector 2

CLDArray	Type	Description
CmdPrm.Int[-15]	Integer	Check for broken tool - Tool number
CmdPrm.Int[-16]	Integer	Check for broken tool - Corrector 1
CmdPrm.Int[-17]	Integer	Check for broken tool - Corrector 2
CmdPrm.Int[-18]	Integer	Calibrate the tool probe number - Probe number
CmdPrm.Int[-19]	Integer	Calibrate the tool probe number - Corrector 1
CmdPrm.Int[-20]	Integer	Calibrate the tool probe number - Corrector 2
CmdPrm.Int[-21]	Integer	Calibrate the part probe number - Probe number
CmdPrm.Int[-22]	Integer	Calibrate the part probe number - Corrector 1
CmdPrm.Int[-23]	Integer	Calibrate the part probe number - Corrector 2
CmdPrm.Int[-24]	Integer	Write to report - Component number
CmdPrm.Int[-25]	Integer	Write to report - Feature number

For `Custom property` element, in `Job assignment` tab of probing cycle operation you manually set the code for the parameter to be stored in CLData and you can also set the type of the parameter. Custom parameter code must not be repeated with other codes.



Probe On/Off

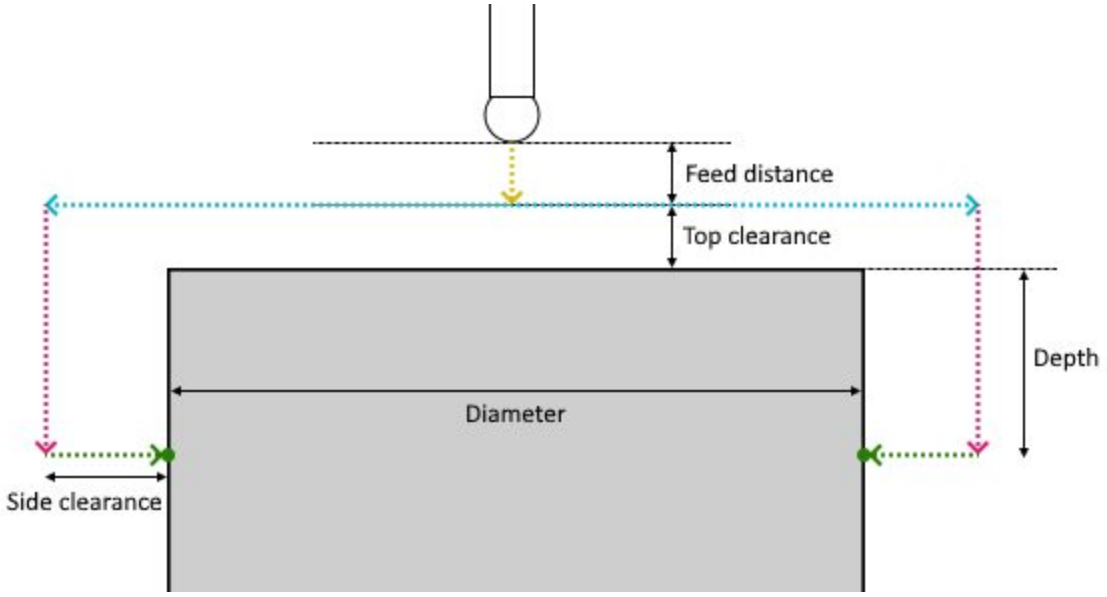
The cycle on and off commands has parameters:

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Cycle On/Off value = 0
CmdPrm.Int[-2]	Integer	Cycle On/Off SubCode = -500
CmdPrm.Bol[-3]	Boolean	On/Off mode. True is "On", false is "Off"

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Boss probing parameters



Boss probing consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting position of the cycle. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance` distance but elevated at the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
6. The tool returns to the starting position;
7. Steps 2-6 are repeated as many times as there are touch points for the cycle;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Boss probing value = 5
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor "

CLDArray	Type	Description
		property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-52]	Double	Diameter of measured boss
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance, approach distance to touch point
CmdPrm.Int[-59]	Integer	Cycle variant. 0 - rectangular mode, 1 - angular mode
CmdPrm.Flt[-60]	Double	Start angle (for cycle variant = 1). Starting angle of touch points
CmdPrm.Flt[-61]	Double	Angular step (for cycle variant = 1). R adial distance between touch points
CmdPrm.Int[-62]	Integer	Step count (for cycle variant = 1). Count of touch points
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis

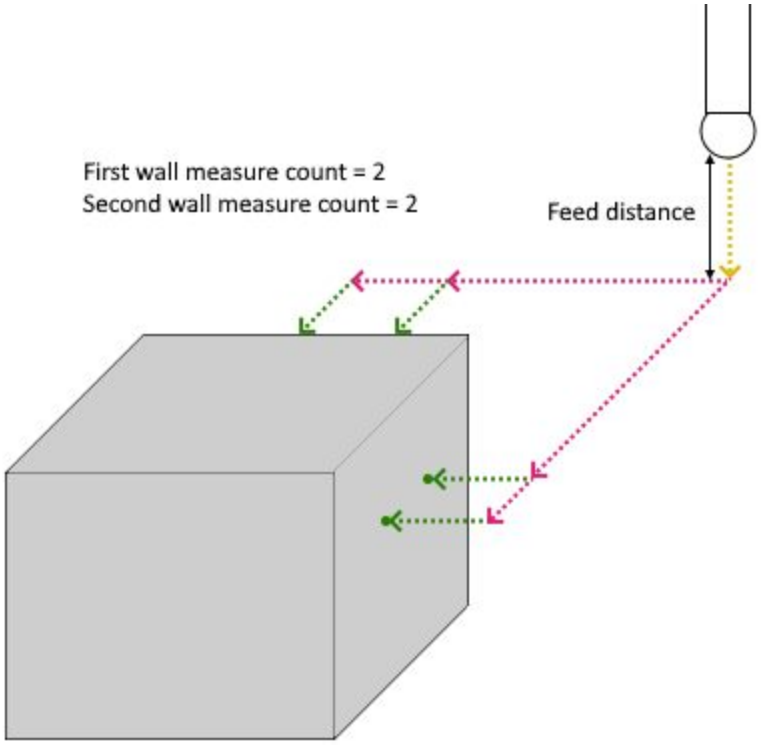
CLDArray	Type	Description
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-100-((N-1)*6)]	Double	Other touch point value along X-axis. N - number of touch point
CmdPrm.Flt[-101-((N-1)*6)]	Double	Other touch point value along Y-axis. N - number of touch point
CmdPrm.Flt[-102-((N-1)*6)]	Double	Other touch point value along Z-axis. N - number of touch point
CmdPrm.Flt[-103-((N-1)*6)]	Double	Other target vector value along X-axis. N - number of touch point
CmdPrm.Flt[-104-((N-1)*6)]	Double	Other target vector value along Y-axis. N - number of touch point
CmdPrm.Flt[-105-((N-1)*6)]	Double	Other target vector value along Z-axis. N - number of touch point
CmdPrm.Flt[-200]	Double	Original first touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	Original first touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original first touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original first target vector value along X-axis specified in the Job

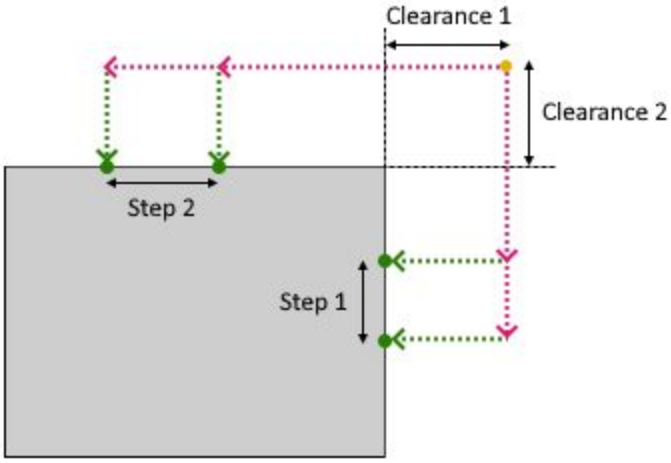
CLDArray	Type	Description
		Assignment
CmdPrm.Flt[-204]	Double	Original first target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original first target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-200-((N-1)*6)]	Double	Other touch point value from original point along X-axis. N - number of touch point
CmdPrm.Flt[-201-((N-1)*6)]	Double	Other touch point value from original point along Y-axis. N - number of touch point
CmdPrm.Flt[-202-((N-1)*6)]	Double	Other touch point value from original point along Z-axis. N - number of touch point
CmdPrm.Flt[-203-((N-1)*6)]	Double	Other target vector value from original vector along X-axis. N - number of touch point
CmdPrm.Flt[-204-((N-1)*6)]	Double	Other target vector value from original vector along Y-axis. N - number of touch point
CmdPrm.Flt[-205-((N-1)*6)]	Double	Other target vector value from original vector along Z-axis. N - number of touch point
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Double wall external corner probing parameters





Double wall external corner probing consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting point of the measurement cycle that is the intersection is the intersection of first touch point of first side and first touch point of second side at a distance specified in `Clearance 1`, `Clearance 2`, respectively, along orthogonal direction of their vectors. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point on first side along its `Target vector` at the `Clearance 1` distance (moving at "Long link feed") and approaches first touch point of the first side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
3. If `First wall measure count` >1 then:
4. The tool moves to the next touch point on the current side at a distance specified in the `Clearance 1` in the direction of its target vector;

5. The tool approaches touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
6. Steps 3 are repeated as many times as indicated in `First wall measure count`;
7. The tool returns to the starting point of the cycle. The movement is executed at "Long link feed";
8. The tool approaches a point remote from the first touch point on second side along its `Target vector` at the `Clearance 2` distance (moving at "Long link feed") approaches first touch point of the second side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
9. If `Second wall measure count` >1 then:
10. The tool moves to the next touch point on the current side at a distance specified in the `Clearance 2` in the direction of its target vector;
11. The tool approaches touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
12. Steps 7 are repeated as many times as indicated in `Second wall measure count`;
13. The tool returns to the starting point of the cycle. Moving at "Long link feed";
14. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Double wall external corner probing value = 14
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-56]	Double	Clearance 1, approach distance to touch points on first side
CmdPrm.Flt[-57]	Double	Clearance 2, approach distance to touch points on second side
CmdPrm.Int[-63]	Integer	First wall measure count, count of touch points on first side

CLDArray	Type	Description
CmdPrm.Int[-64]	Integer	Second wall measure count, count of touch points on second side
CmdPrm.Flt[-65]	Double	Step 1, distance between touch points on first wall
CmdPrm.Flt[-67]	Double	Step 2, distance between touch points on second wall
CmdPrm.Flt[-100]	Double	First touch point on first side value along X-axis
CmdPrm.Flt[-101]	Double	First touch point on first side value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point on first side value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector on first side value along X-axis
CmdPrm.Flt[-104]	Double	First target vector on first side value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector on first side value along Z-axis
CmdPrm.Flt[-100-((N-1)*6)]	Double	Other touch point on first/second side value along X-axis. N - number of touch point
CmdPrm.Flt[-101-((N-1)*6)]	Double	Other touch point on first/second side value along Y-axis. N - number of touch point
CmdPrm.Flt[-102-((N-1)*6)]	Double	Other touch point on first/second side value along Z-axis. N - number of touch point
CmdPrm.Flt[-103-((N-1)*6)]	Double	Other target vector on first/second side value along X-

CLDArray	Type	Description
		axis. N - number of touch point
CmdPrm.Flt[-104-((N-1)*6)]	Double	Other target vector on first/second side value along Y-axis. N - number of touch point
CmdPrm.Flt[-105-((N-1)*6)]	Double	Other target vector on first/second side value along Z-axis. N - number of touch point
CmdPrm.Flt[-100-6*C]	Double	First touch point on second side value along X-axis. C - count of touch point on first side
CmdPrm.Flt[-101-6*C]	Double	First touch point on second side value along Y-axis. C - count of touch point on first side
CmdPrm.Flt[-102-6*C]	Double	First touch point on second side value along Z-axis. C - count of touch point on first side
CmdPrm.Flt[-103-6*C]	Double	First target vector on second side value along X-axis. C - count of touch point on first side
CmdPrm.Flt[-104-6*C]	Double	First target vector on second side value along Y-axis. C - count of touch point on first side
CmdPrm.Flt[-105-6*C]	Double	First target vector on second side value along Z-axis. C - count of touch point on first side
CmdPrm.Flt[-100-6*C-((N-1)*6)]	Double	Other touch point on second side value along X-axis. N - number of touch point. C - count of touch point on first side
CmdPrm.Flt[-101-6*C-((N-1)*6)]	Double	Other touch point on second side value along Y-axis. N - number of touch point on

CLDArray	Type	Description
		second side. C - count of touch point on first side
CmdPrm.Flt[-102-6*C-((N-1)*6)]	Double	Other touch point on second side value along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-103-6*C-((N-1)*6)]	Double	Other target vector on second side value along X-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-104-6*C-((N-1)*6)]	Double	Other target vector on second side value along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-105-6*C-((N-1)*6)]	Double	Other target vector on second side value along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-200]	Double	Original first touch point on first side value along X-axis
CmdPrm.Flt[-201]	Double	Original first touch point on first side value along Y-axis
CmdPrm.Flt[-202]	Double	Original first touch point on first side value along Z-axis
CmdPrm.Flt[-203]	Double	Original first target vector on first side value along X-axis
CmdPrm.Flt[-204]	Double	Original first target vector on first side value along Y-axis

CLDArray	Type	Description
CmdPrm.Flt[-205]	Double	Original first target vector on first side value along Z-axis
CmdPrm.Flt[-200-((N-1)*6)]	Double	Other touch point on first/second side value from original point along X-axis. N - number of touch point
CmdPrm.Flt[-201-((N-1)*6)]	Double	Other touch point on first/second side value from original point along Y-axis. N - number of touch point
CmdPrm.Flt[-202-((N-1)*6)]	Double	Other touch point on first/second side value from original point along Z-axis. N - number of touch point
CmdPrm.Flt[-203-((N-1)*6)]	Double	Other target vector on first/second side value from original vector along X-axis. N - number of touch point
CmdPrm.Flt[-204-((N-1)*6)]	Double	Other target vector on first/second side value from original vector along Y-axis. N - number of touch point
CmdPrm.Flt[-205-((N-1)*6)]	Double	Other target vector on first/second side value from original vector along Z-axis. N - number of touch point
CmdPrm.Flt[-200-6*C]	Double	First touch point on second side value along X-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-201-6*C]	Double	First touch point on second side value along Y-axis specified in

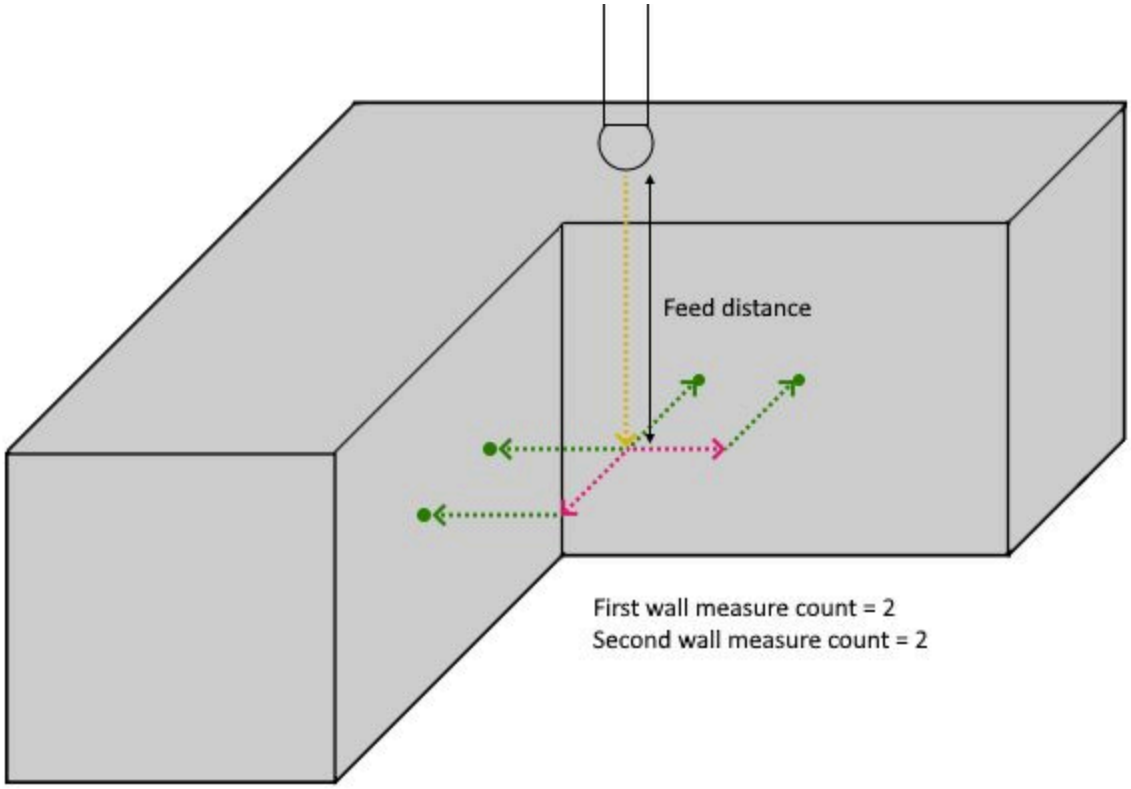
CLDArray	Type	Description
		the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-202-6*C]	Double	First touch point on second side value along Z-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-203-6*C]	Double	First target vector on second side value along X-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-204-6*C]	Double	First target vector on second side value along Y-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-205-6*C]	Double	First target vector on second side value along Z-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-200-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along X-axis. N - number of touch point. C - count of touch point on first side
CmdPrm.Flt[-201-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-202-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along Z-axis. N - number of touch point on second side. C - count of touch point on first side

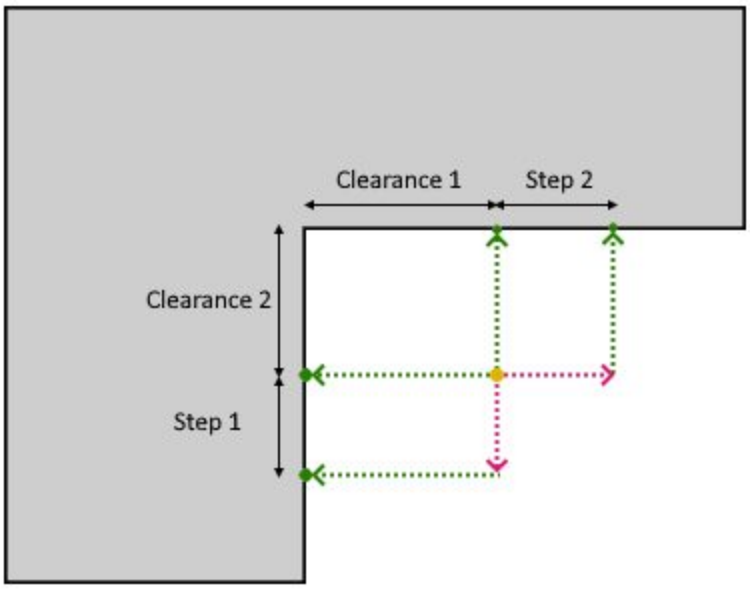
CLDArray	Type	Description
CmdPrm.Flt[-203-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along X-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-204-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-205-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Double wall internal corner probing parameters





Double wall internal corner probing consist of the following steps:

1. The tool travels to the distance specified in the `Feed distance` and approaches the starting point of the measurement cycle that is the intersection of first touch point of first side and first touch point of second side at a distance specified in `Clearance 1`, `Clearance 2`, respectively, along their vectors. Moving at "Approach feed";

2. The tool approaches first touch point of the first side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
3. If `First wall measure count` >1 then:
4. The tool moves to the next touch point on the current side at a distance specified in the `Clearance 1` in the direction of its target vector;
5. The tool approaches touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
6. Steps 3 are repeated as many times as indicated in `First wall measure count`;
7. The tool returns to the starting point of the cycle. Moving at "Long link feed";
8. The tool approaches first touch point of the second side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
9. If `Second wall measure count` >1 then:
10. The tool moves to the next touch point on the current side at a distance specified in the `Clearance 2` in the direction of its target vector;
11. The tool approaches touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
12. Steps 7 are repeated as many times as indicated in `Second wall measure count`;
13. The tool returns to the starting point of the cycle. Moving at "Long link feed";
14. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Double wall internal corner probing value = 13
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-56]	Double	Clearance 1, approach distance to touch points on first side
CmdPrm.Flt[-57]	Double	Clearance 2, approach distance to touch points on second side

CLDArray	Type	Description
CmdPrm.Int[-63]	Integer	First wall measure count, count of touch points on first side
CmdPrm.Int[-64]	Integer	Second wall measure count, count of touch points on second side
CmdPrm.Flt[-65]	Double	Step 1, distance between touch points on first wall
CmdPrm.Flt[-67]	Double	Step 2, distance between touch points on second wall
CmdPrm.Flt[-100]	Double	First touch point on first side value along X-axis
CmdPrm.Flt[-101]	Double	First touch point on first side value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point on first side value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector on first side value along X-axis
CmdPrm.Flt[-104]	Double	First target vector on first side value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector on first side value along Z-axis
CmdPrm.Flt[-100-((N-1)*6)]	Double	Other touch point on first side value along X-axis. N - number of touch point on first side
CmdPrm.Flt[-101-((N-1)*6)]	Double	Other touch point on first side value along Y-axis. N - number of touch point on first side

CLDArray	Type	Description
CmdPrm.Flt[-102-((N-1)*6)]	Double	Other touch point on first side value along Z-axis. N - number of touch point on first side
CmdPrm.Flt[-103-((N-1)*6)]	Double	Other target vector on first side value along X-axis. N - number of touch point on first side
CmdPrm.Flt[-104-((N-1)*6)]	Double	Other target vector on first side value along Y-axis. N - number of touch point on first side
CmdPrm.Flt[-105-((N-1)*6)]	Double	Other target vector on first side value along Z-axis. N - number of touch point on first side
CmdPrm.Flt[-100-6*C]	Double	First touch point on second side value along X-axis. C - count of touch point on first side
CmdPrm.Flt[-101-6*C]	Double	First touch point on second side value along Y-axis. C - count of touch point on first side
CmdPrm.Flt[-102-6*C]	Double	First touch point on second side value along Z-axis. C - count of touch point on first side
CmdPrm.Flt[-103-6*C]	Double	First target vector on second side value along X-axis. C - count of touch point on first side
CmdPrm.Flt[-104-6*C]	Double	First target vector on second side value along Y-axis. C - count of touch point on first side
CmdPrm.Flt[-105-6*C]	Double	First target vector on second side value along Z-axis. C - count of touch point on first side

CLDArray	Type	Description
CmdPrm.Flt[-100-6*C-((N-1)*6)]	Double	Other touch point on second side value along X-axis. N - number of touch point. C - count of touch point on first side
CmdPrm.Flt[-101-6*C-((N-1)*6)]	Double	Other touch point on second side value along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-102-6*C-((N-1)*6)]	Double	Other touch point on second side value along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-103-6*C-((N-1)*6)]	Double	Other target vector on second side value along X-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-104-6*C-((N-1)*6)]	Double	Other target vector on second side value along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-105-6*C-((N-1)*6)]	Double	Other target vector on second side value along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-200]	Double	Original first touch point on first side value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-201]	Double	Original first touch point on first side value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original first touch point on first side value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original first target vector on first side value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	Original first target vector on first side value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original first target vector on first side value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-200-((N-1)*6)]	Double	Other touch point on first side value from original point along X-axis. N - number of touch point on first side
CmdPrm.Flt[-201-((N-1)*6)]	Double	Other touch point on first side value from original point along Y-axis. N - number of touch point on first side
CmdPrm.Flt[-202-((N-1)*6)]	Double	Other touch point on first side value from original point along Z-axis. N - number of touch point on first side
CmdPrm.Flt[-203-((N-1)*6)]	Double	Other target vector on first side value from original vector along X-axis. N - number of touch point on first side

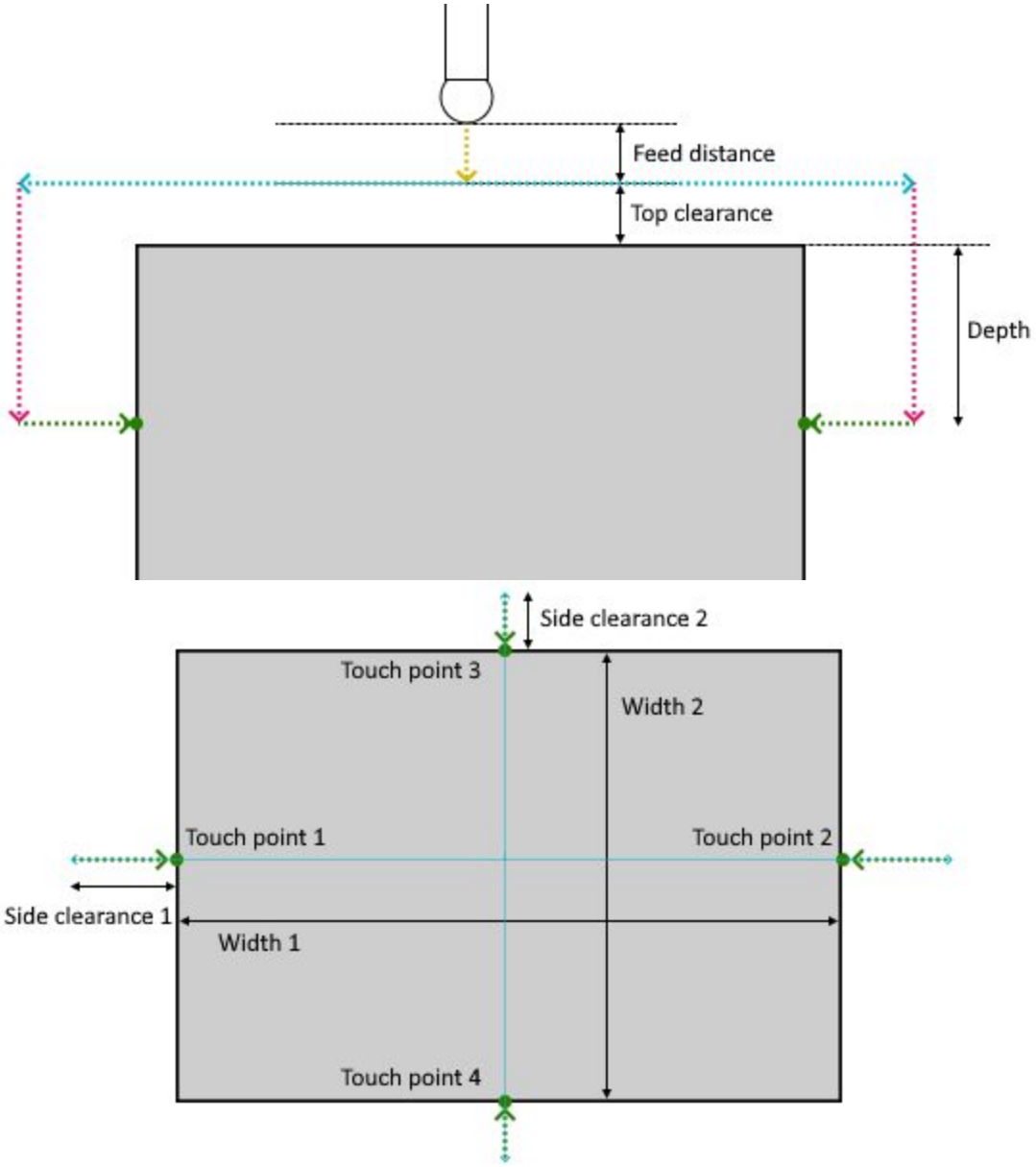
CLDArray	Type	Description
CmdPrm.Flt[-204-((N-1)*6)]	Double	Other target vector on first side value from original vector along Y-axis. N - number of touch point on first side
CmdPrm.Flt[-205-((N-1)*6)]	Double	Other target vector on first side value from original vector along Z-axis. N - number of touch point on first side
CmdPrm.Flt[-200-6*C]	Double	First touch point on second side value along X-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-201-6*C]	Double	First touch point on second side value along Y-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-202-6*C]	Double	First touch point on second side value along Z-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-203-6*C]	Double	First target vector on second side value along X-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-204-6*C]	Double	First target vector on second side value along Y-axis specified in the Job Assignment. C - count of touch point on first side
CmdPrm.Flt[-205-6*C]	Double	First target vector on second side value along Z-axis specified in the Job Assignment. C - count of touch point on first side

CLDArray	Type	Description
CmdPrm.Flt[-200-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along X-axis. N - number of touch point. C - count of touch point on first side
CmdPrm.Flt[-201-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-202-6*C-((N-1)*6)]	Double	Other touch point on second side value from original point along Z-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-203-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along X-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-204-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along Y-axis. N - number of touch point on second side. C - count of touch point on first side
CmdPrm.Flt[-205-6*C-((N-1)*6)]	Double	Other target vector on second side value from original vector along Z-axis. N - number of touch point on second side. C - count of touch point on first side

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

External rectangle probing parameters



External rectangle probing consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting position of the cycle. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance 1` distance but elevated at the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";

6. The tool returns to the starting position;
7. Steps 2-6 are repeated 3 more times for each touch points;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: External rectangle probing value = 8
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width 1, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-54]	Double	Width 2, distance from Touch point 3 to Touch point 4
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance 1, approach distance to touch point 1/ touch point 2
CmdPrm.Flt[-57]	Double	Side clearance 2, approach distance to touch point 3/ touch point 4
CmdPrm.Flt[-100]	Double	First touch point value along X-axis

CLDArray	Type	Description
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-112]	Double	Third touch point value along X-axis
CmdPrm.Flt[-113]	Double	Third touch point value along Y-axis
CmdPrm.Flt[-114]	Double	Third touch point value along Z-axis

CLDArray	Type	Description
CmdPrm.Flt[-115]	Double	Third target vector value along X-axis
CmdPrm.Flt[-116]	Double	Third target vector value along Y-axis
CmdPrm.Flt[-117]	Double	Third target vector value along Z-axis
CmdPrm.Flt[-118]	Double	Fourth touch point value along X-axis
CmdPrm.Flt[-119]	Double	Fourth touch point value along Y-axis
CmdPrm.Flt[-120]	Double	Fourth touch point value along Z-axis
CmdPrm.Flt[-121]	Double	Fourth target vector value along X-axis
CmdPrm.Flt[-122]	Double	Fourth target vector value along Y-axis
CmdPrm.Flt[-123]	Double	Fourth target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-212]	Double	Third original touch point value along X-axis specified in the Job Assignment

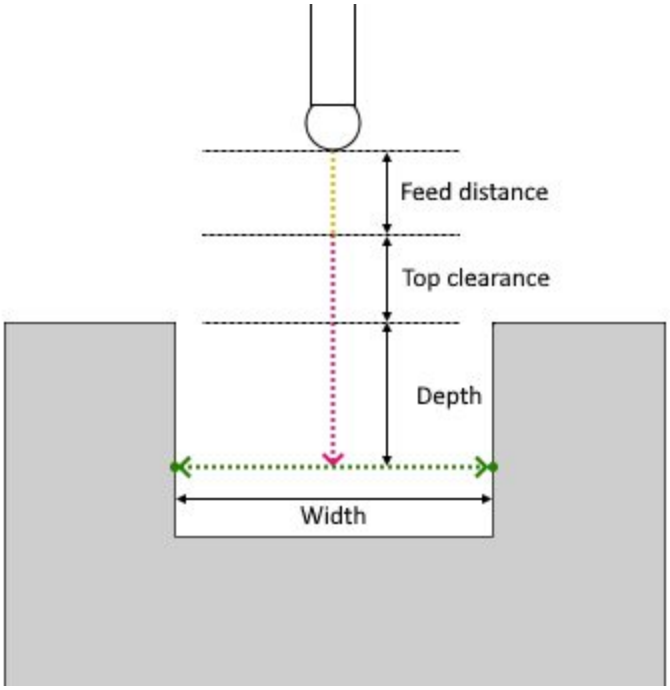
CLDArray	Type	Description
CmdPrm.Flt[-213]	Double	Third original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Third original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Third original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Third original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-217]	Double	Third original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-218]	Double	Fourth original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-219]	Double	Fourth original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-220]	Double	Fourth original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-221]	Double	Fourth original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-222]	Double	Fourth original target vector value along Y-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-223]	Double	Fourth original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Groove probing parameters



Groove probing consist of the following steps:

1. If the `Top clearance` parameter is checked:
2. The tool moves to the distance specified in the `Feed distance` and approaches the starting point of the `Top clearance` position. The movement is executed at "Approach feed";
3. The tool moves to the distance specified in the `Top clearance` plus `Depth` value and approaches the center of touch points (starting point of cycle). The movement is executed at "Long link feed";
4. If the `Top clearance` parameter is unchecked:
5. The tool moves to the distance specified in the `Feed distance` and approaches the center of touch points (starting point of cycle) according to depth. The movement is executed at "Approach feed";
6. The tool approaches the first touch point (moving at "Work feed") along its inverted `Target vector` and then returns (moving at "Long link feed") to the center point. This action is repeated for the second touch point;
7. If the `Top clearance` parameter is checked: the tool moves up a distance equal to the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
8. The tool retracts a distance to the `Feed distance` position. The movement is executed at "Return feed".

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Groove probing value = 11

CLDArray	Type	Description
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis

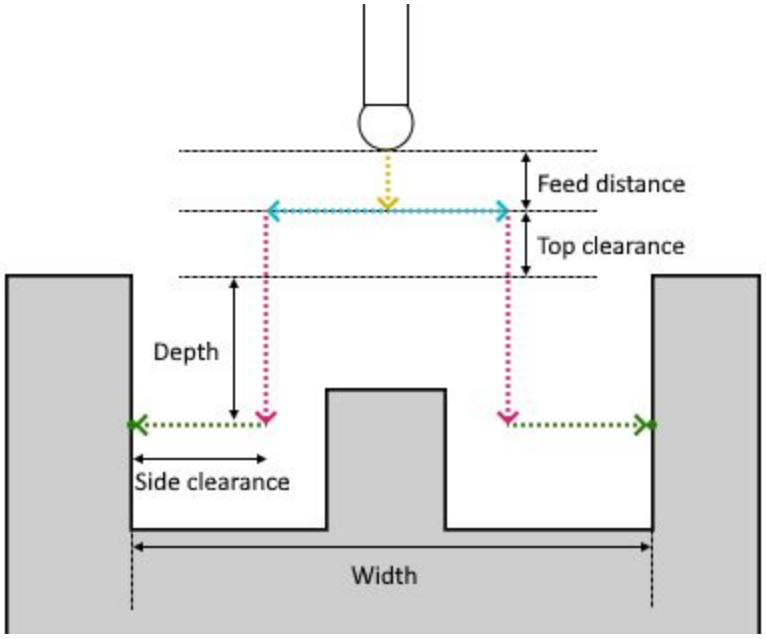
CLDArray	Type	Description
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Groove probing protected parameters



Groove probing protected consist of the following steps:

1. The tool moves to the distance specified in the `Feed distance` and approaches the starting position of the cycle. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance` distance but elevated at the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
6. The tool returns to the starting position;
7. Steps 2-6 are repeated for second touch point;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Groove probing protected value = 12
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor "

CLDArray	Type	Description
		property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance, approach distance to touch point 1/ touch point 2
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis

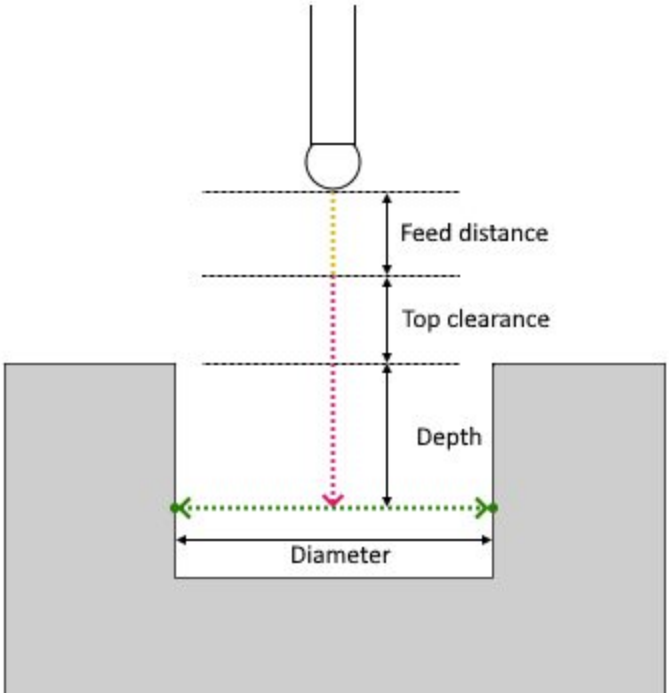
CLDArray	Type	Description
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Hole probing parameters



Hole probing consist of the following steps:

1. If the `Top clearance` parameter is checked:
2. The tool moves to the distance specified in the `Feed distance` and approaches the starting point of the `Top clearance` position. The movement is executed at "Approach feed";
3. The tool moves to the distance specified in the `Top clearance` plus `Depth` value and approaches the center of hole (starting point of cycle). The movement is executed at "Long link feed";
4. If the `Top clearance` parameter is unchecked:
5. The tool moves to the distance specified in the `Feed distance` and approaches the center of hole (starting point of cycle) according to depth. The movement is executed at "Approach feed";
6. The tool approaches the first touch point (moving at "Work feed") along its inverted `Target vector` and then returns (moving at "Long link feed") to the center point. This action is repeated as many times as there are touch points;
7. If the `Top clearance` parameter is checked: the tool moves up a distance equal to the `Top clearance` value + `Depth` value. Moving at "Long link feed";
8. The tool retracts a distance to the `Feed distance` position. Moving at "Return feed".

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Hole probing value = 3

CLDArray	Type	Description
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-52]	Double	Diameter of measured hole
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Int[-59]	Integer	Cycle variant. 0 - rectangular mode, 1 - angular mode
CmdPrm.Flt[-60]	Double	Start angle (for cycle variant = 1). Starting angle of touch points
CmdPrm.Flt[-61]	Double	Angular step (for cycle variant = 1). R adial distance between touch points
CmdPrm.Int[-62]	Integer	Step count (for cycle variant = 1). Count of touch points
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis

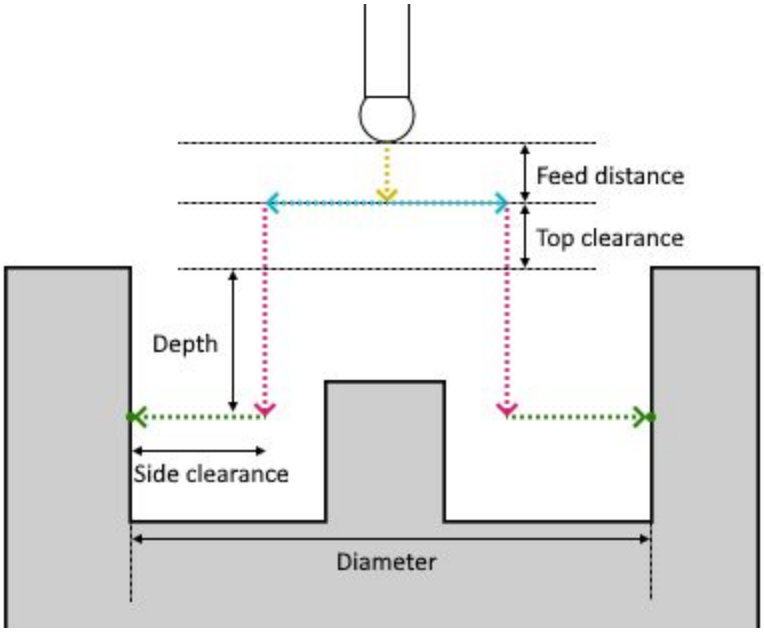
CLDArray	Type	Description
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-100-((N-1)*6)]	Double	Other touch point value along X-axis. N - number of touch point
CmdPrm.Flt[-101-((N-1)*6)]	Double	Other touch point value along Y-axis. N - number of touch point
CmdPrm.Flt[-102-((N-1)*6)]	Double	Other touch point value along Z-axis. N - number of touch point
CmdPrm.Flt[-103-((N-1)*6)]	Double	Other target vector value along X-axis. N - number of touch point
CmdPrm.Flt[-104-((N-1)*6)]	Double	Other target vector value along Y-axis. N - number of touch point
CmdPrm.Flt[-105-((N-1)*6)]	Double	Other target vector value along Z-axis. N - number of touch point
CmdPrm.Flt[-200]	Double	Original first touch point value along X-axis specified in theJob Assignment
CmdPrm.Flt[-201]	Double	Original first touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original first touch point value along Z-axis specified in theJob Assignment
CmdPrm.Flt[-203]	Double	Original first target vector value along X-axis specified in the Job

CLDArray	Type	Description
		Assignment
CmdPrm.Flt[-204]	Double	Original first target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original first target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-200-((N-1)*6)]	Double	Other touch point value from original point along X-axis. N - number of touch point
CmdPrm.Flt[-201-((N-1)*6)]	Double	Other touch point value from original point along Y-axis. N - number of touch point
CmdPrm.Flt[-202-((N-1)*6)]	Double	Other touch point value from original point along Z-axis. N - number of touch point
CmdPrm.Flt[-203-((N-1)*6)]	Double	Other target vector value from original vector along X-axis. N - number of touch point
CmdPrm.Flt[-204-((N-1)*6)]	Double	Other target vector value from original vector along Y-axis. N - number of touch point
CmdPrm.Flt[-205-((N-1)*6)]	Double	Other target vector value from original vector along Z-axis. N - number of touch point

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Hole protected probing parameters



Hole protected probing consist of the following steps:

1. The tool moves to the distance specified in the `Feed distance` and approaches the starting position of the cycle. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance` distance but elevated at the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
6. The tool returns to the starting position;
7. Steps 2-6 are repeated as many times as there are touch points for the cycle;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Hole protected probing value = 4
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor "

CLDArray	Type	Description
		property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-52]	Double	Diameter of measured hole
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance, approach distance to touch point
CmdPrm.Int[-59]	Integer	Cycle variant. 0 - rectangular mode, 1 - angular mode
CmdPrm.Flt[-60]	Double	Start angle (for cycle variant = 1). Starting angle of touch points
CmdPrm.Flt[-61]	Double	Angular step (for cycle variant = 1). R adial distance between touch points
CmdPrm.Int[-62]	Integer	Step count (for cycle variant = 1). Count of touch points
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis

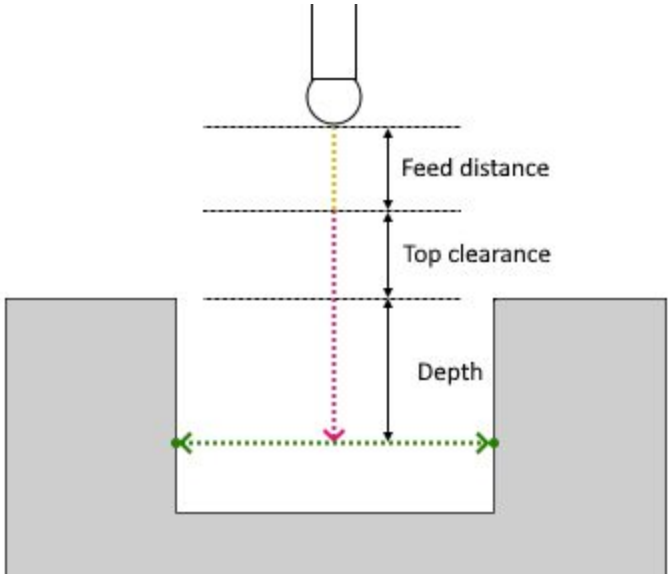
CLDArray	Type	Description
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-100-((N-1)*6)]	Double	Other touch point value along X-axis. N - number of touch point
CmdPrm.Flt[-101-((N-1)*6)]	Double	Other touch point value along Y-axis. N - number of touch point
CmdPrm.Flt[-102-((N-1)*6)]	Double	Other touch point value along Z-axis. N - number of touch point
CmdPrm.Flt[-103-((N-1)*6)]	Double	Other target vector value along X-axis. N - number of touch point
CmdPrm.Flt[-104-((N-1)*6)]	Double	Other target vector value along Y-axis. N - number of touch point
CmdPrm.Flt[-105-((N-1)*6)]	Double	Other target vector value along Z-axis. N - number of touch point
CmdPrm.Flt[-200]	Double	Original first touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	Original first touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original first touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original first target vector value along X-axis specified in the Job

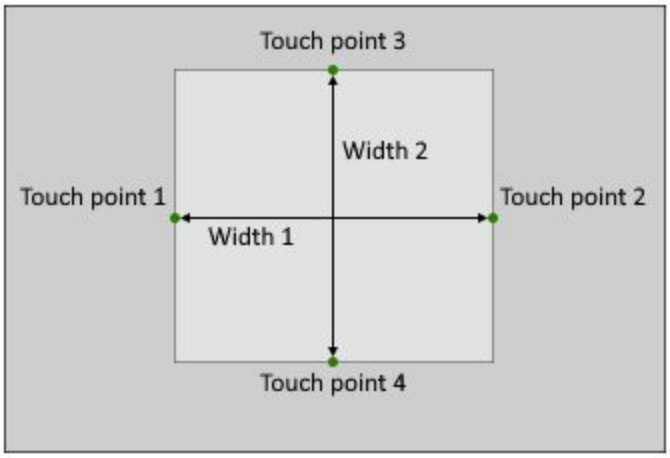
CLDArray	Type	Description
		Assignment
CmdPrm.Flt[-204]	Double	Original first target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original first target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-200-((N-1)*6)]	Double	Other touch point value from original point along X-axis. N - number of touch point
CmdPrm.Flt[-201-((N-1)*6)]	Double	Other touch point value from original point along Y-axis. N - number of touch point
CmdPrm.Flt[-202-((N-1)*6)]	Double	Other touch point value from original point along Z-axis. N - number of touch point
CmdPrm.Flt[-203-((N-1)*6)]	Double	Other target vector value from original vector along X-axis. N - number of touch point
CmdPrm.Flt[-204-((N-1)*6)]	Double	Other target vector value from original vector along Y-axis. N - number of touch point
CmdPrm.Flt[-205-((N-1)*6)]	Double	Other target vector value from original vector along Z-axis. N - number of touch point
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Internal rectangle probing parameters





Internal rectangle probing consist of the following steps:

1. If the `Top clearance` parameter is checked:
2. The tool moves to the distance specified in the `Feed distance` and approaches the starting point of the `Top clearance` position. The movement is executed at "Approach feed";
3. The tool moves to the distance specified in the `Top clearance` plus `Depth` value and approaches the center of sides (starting point of cycle). The movement is executed at "Long link feed";
4. If the `Top clearance` parameter is unchecked:
5. The tool moves to the distance specified in the `Feed distance` and approaches the center of sides (starting point of cycle) according to depth. Moving at "Approach feed";
6. The tool approaches the first touch point (moving at "Work feed") along its inverted `Target vector` and then returns (moving at "Long link feed") to the center point. This action is repeated 3 times for each touch points;
7. If the `Top clearance` parameter is checked: the tool moves up a distance equal to the `Top clearance` value + `Depth` value. Moving at "Long link feed";
8. The tool retracts a distance to the `Feed distance` position. Moving at "Return feed".

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Internal rectangle probing value = 6
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor "

CLDArray	Type	Description
		property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width 1, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-54]	Double	Width 2, distance from Touch point 3 to Touch point 4
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis

CLDArray	Type	Description
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-112]	Double	Third touch point value along X-axis
CmdPrm.Flt[-113]	Double	Third touch point value along Y-axis
CmdPrm.Flt[-114]	Double	Third touch point value along Z-axis
CmdPrm.Flt[-115]	Double	Third target vector value along X-axis
CmdPrm.Flt[-116]	Double	Third target vector value along Y-axis
CmdPrm.Flt[-117]	Double	Third target vector value along Z-axis
CmdPrm.Flt[-118]	Double	Fourth touch point value along X-axis
CmdPrm.Flt[-119]	Double	Fourth touch point value along Y-axis
CmdPrm.Flt[-120]	Double	Fourth touch point value along Z-axis
CmdPrm.Flt[-121]	Double	Fourth target vector value along X-axis

CLDArray	Type	Description
CmdPrm.Flt[-122]	Double	Fourth target vector value along Y-axis
CmdPrm.Flt[-123]	Double	Fourth target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment

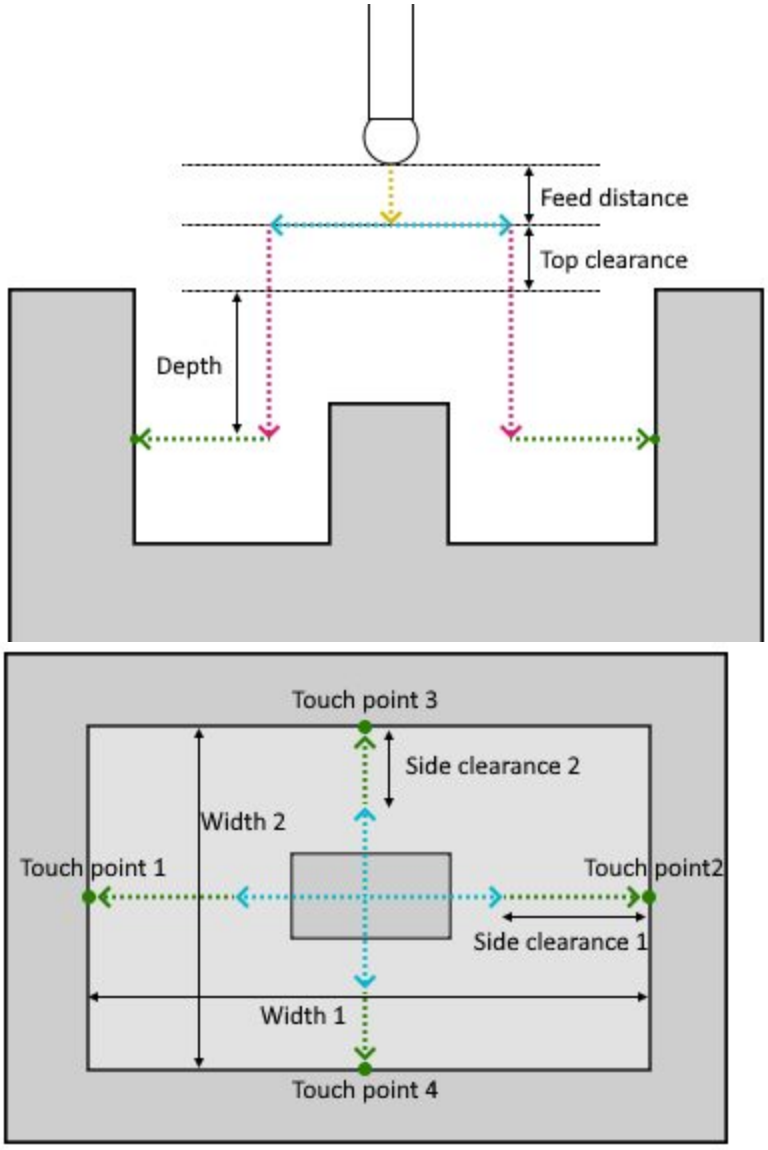
CLDArray	Type	Description
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-212]	Double	Third original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-213]	Double	Third original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Third original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Third original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Third original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-217]	Double	Third original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-218]	Double	Fourth original touch point value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-219]	Double	Fourth original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-220]	Double	Fourth original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-221]	Double	Fourth original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-222]	Double	Fourth original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-223]	Double	Fourth original target vector value along Z-axis specified in the Job Assignment

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Internal rectangle probing protected parameters



Internal rectangle probing protected consist of the following steps:

1. The tool moves to the distance specified in the `Feed distance` and approaches the starting position of the cycle. The movement is executed at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance 1` distance but elevated at the `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. Moving at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. The movement is executed at "Long link feed";
6. The tool returns to the starting position;

7. Steps 2-6 are repeated 3 times for each touch points;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Internal rectangle probing protected value = 7
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width 1, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-54]	Double	Width 2, distance from Touch point 3 to Touch point 4
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance 1, approach distance to touch point 1/ touch point 2
CmdPrm.Flt[-57]	Double	Side clearance 2, approach distance to touch point 3/ touch point 4
CmdPrm.Flt[-100]	Double	First touch point value along X-axis

CLDArray	Type	Description
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-112]	Double	Third touch point value along X-axis
CmdPrm.Flt[-113]	Double	Third touch point value along Y-axis
CmdPrm.Flt[-114]	Double	Third touch point value along Z-axis

CLDArray	Type	Description
CmdPrm.Flt[-115]	Double	Third target vector value along X-axis
CmdPrm.Flt[-116]	Double	Third target vector value along Y-axis
CmdPrm.Flt[-117]	Double	Third target vector value along Z-axis
CmdPrm.Flt[-118]	Double	Fourth touch point value along X-axis
CmdPrm.Flt[-119]	Double	Fourth touch point value along Y-axis
CmdPrm.Flt[-120]	Double	Fourth touch point value along Z-axis
CmdPrm.Flt[-121]	Double	Fourth target vector value along X-axis
CmdPrm.Flt[-122]	Double	Fourth target vector value along Y-axis
CmdPrm.Flt[-123]	Double	Fourth target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-212]	Double	Third original touch point value along X-axis specified in the Job Assignment

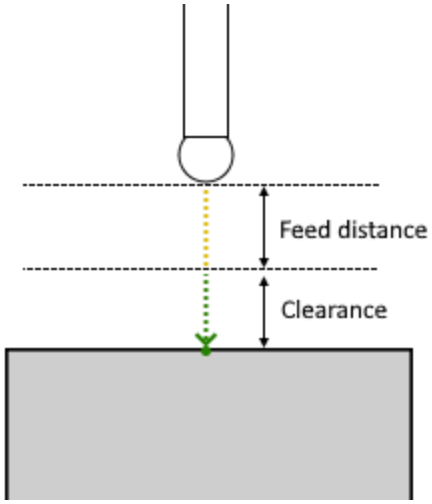
CLDArray	Type	Description
CmdPrm.Flt[-213]	Double	Third original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Third original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Third original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Third original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-217]	Double	Third original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-218]	Double	Fourth original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-219]	Double	Fourth original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-220]	Double	Fourth original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-221]	Double	Fourth original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-222]	Double	Fourth original target vector value along Y-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-223]	Double	Fourth original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Surface probing parameters



Surface probing consist of the following steps:

1. The tool moves to the distance specified in the `Feed distance` and approaches the starting point of the measurement cycle. The movement is executed at "Approach feed";
2. The tool moves to the distance specified in the `Clearance` and approaches touch point of the surface. The movement is executed at "Work feed";
3. The tool returns to starting point for a distance specified in the `Clearance`. The movement is executed at "Retract feed";
4. The tool returns for a distance specified in the `Feed distance`. The movement is executed at "Return feed".

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Surface probing value = 2
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in "SubCode for postprocessor" property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-100]	Double	Touch point value along X-axis
CmdPrm.Flt[-101]	Double	T ouch point value along Y-axis

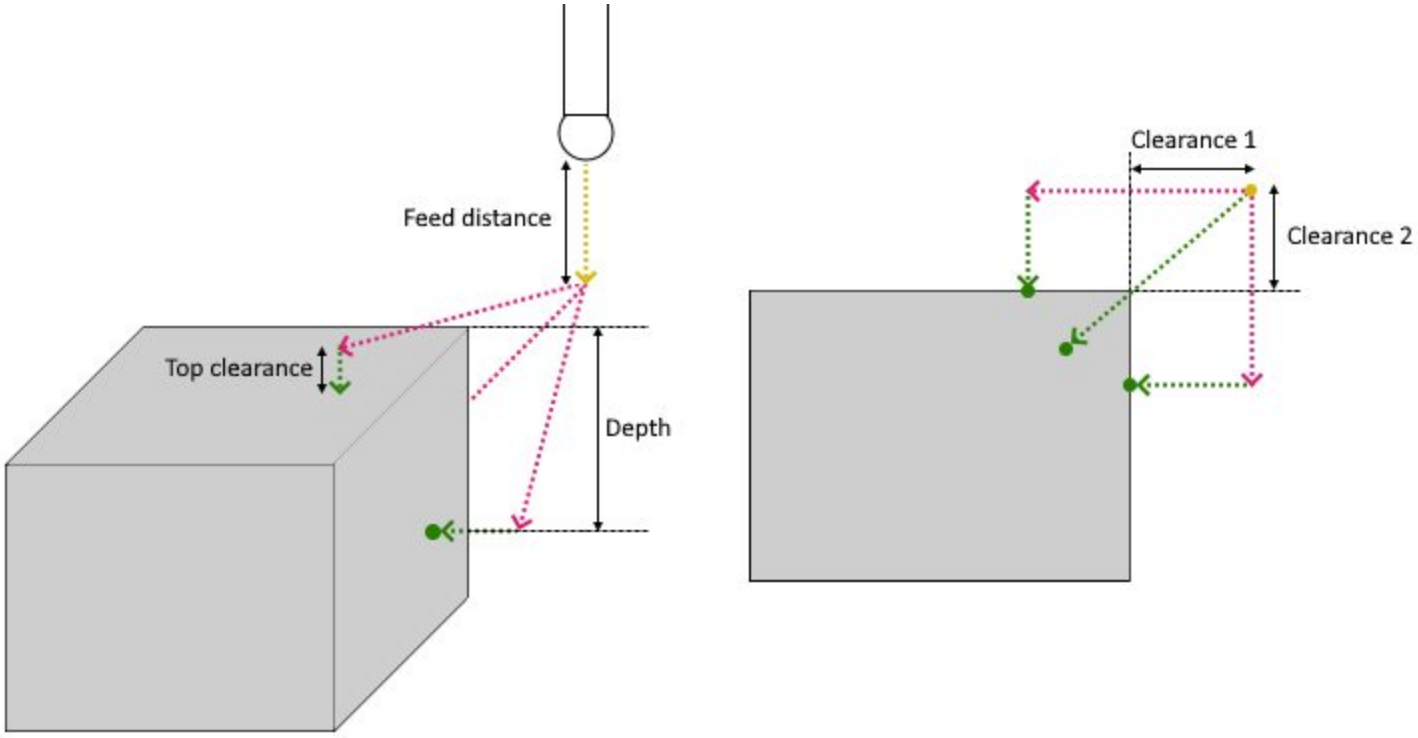
CLDArray	Type	Description
CmdPrm.Flt[-102]	Double	T ouch point value along Z-axis
CmdPrm.Flt[-103]	Double	Target vector value along X-axis
CmdPrm.Flt[-104]	Double	Target vector value along Y-axis
CmdPrm.Flt[-105]	Double	Target vector value along Z-axis
CmdPrm.Flt[-200]	Double	Original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	Original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	Original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-56]	Double	Clearance, approach distance to touch point
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)

- CLData access model

Triple wall external corner probing parameters



Triple wall external corner probing consists of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting point of the measurement cycle that is the intersection of touch point on first wall and touch point on second wall at a distance specified in `Clearance 1`, `Clearance 2`, respectively, along orthogonal direction of their vectors and the intersection point is raised by the distance specified in the `Top clearance` + `Depth`. Moving at "Approach feed";
2. The tool approaches a point remote from touch point on first side along its `Target vector` at the `Clearance 1` distance (moving at "Long link feed") and approaches touch point of the first side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
3. The tool returns to the starting point. Moving at "Long link feed";
4. The tool approaches a point remote from touch point on second side along its `Target vector` at the `Clearance 2` distance (moving at "Long link feed") and approaches touch point of the second side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool returns to the starting point. Moving at "Long link feed";
6. The tool approaches a point remote from touch point on top side along its `Target vector` at the `Top clearance` distance (moving at "Long link feed") and approaches touch point of the top side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
7. The tool returns to the starting point. Moving at "Long link feed";
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Triple wall external corner probing value = 16
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to touch point of fisrt and second wall
CmdPrm.Flt[-56]	Double	Clearance 1, approach distance to touch point on first side
CmdPrm.Flt[-57]	Double	Clearance 2, approach distance to touch point on second side
CmdPrm.Flt[-58]	Double	Top clearance, approach distance to touch point on top side
CmdPrm.Flt[-100]	Double	Touch point on first wall value along X-axis
CmdPrm.Flt[-101]	Double	T ouch point on first wall value along Y-axis
CmdPrm.Flt[-102]	Double	T ouch point on first wall value along Z-axis
CmdPrm.Flt[-103]	Double	T arget vector of touch point on first wall value along X-axis
CmdPrm.Flt[-104]	Double	T arget vector of touch point on first wall value along Y-axis

CLDArray	Type	Description
CmdPrm.Flt[-105]	Double	T arget vector of touch point on first wall value along Z-axis
CmdPrm.Flt[-106]	Double	T ouch point on second wall value along X-axis
CmdPrm.Flt[-107]	Double	T ouch point on second wall value along Y-axis
CmdPrm.Flt[-108]	Double	T ouch point on second wall value along Z-axis
CmdPrm.Flt[-109]	Double	T arget vector of touch point on second wall value along X-axis
CmdPrm.Flt[-110]	Double	T arget vector of touch point on second wall value along Y-axis
CmdPrm.Flt[-111]	Double	T arget vector of touch point on second wall value along Z-axis
CmdPrm.Flt[-112]	Double	T ouch point on top wall value along X-axis
CmdPrm.Flt[-113]	Double	T ouch point on top wall value along Y-axis
CmdPrm.Flt[-114]	Double	T ouch point on top wall value along Z-axis
CmdPrm.Flt[-115]	Double	T arget vector of touch point on top wall value along X-axis
CmdPrm.Flt[-116]	Double	T arget vector of touch point on top wall value along Y-axis
CmdPrm.Flt[-117]	Double	T arget vector of touch point on top wall value along Z-axis
CmdPrm.Flt[-200]	Double	Original touch point on first wall value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-201]	Double	Original touch point on first wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original touch point on first wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original target vector of touch point on first wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	Original target vector of touch point on first wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original target vector of touch point on first wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Original touch point on second wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Original touch point on second wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Original touch point on second wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Original target vector of touch point on second wall value along X-axis specified in the Job Assignment

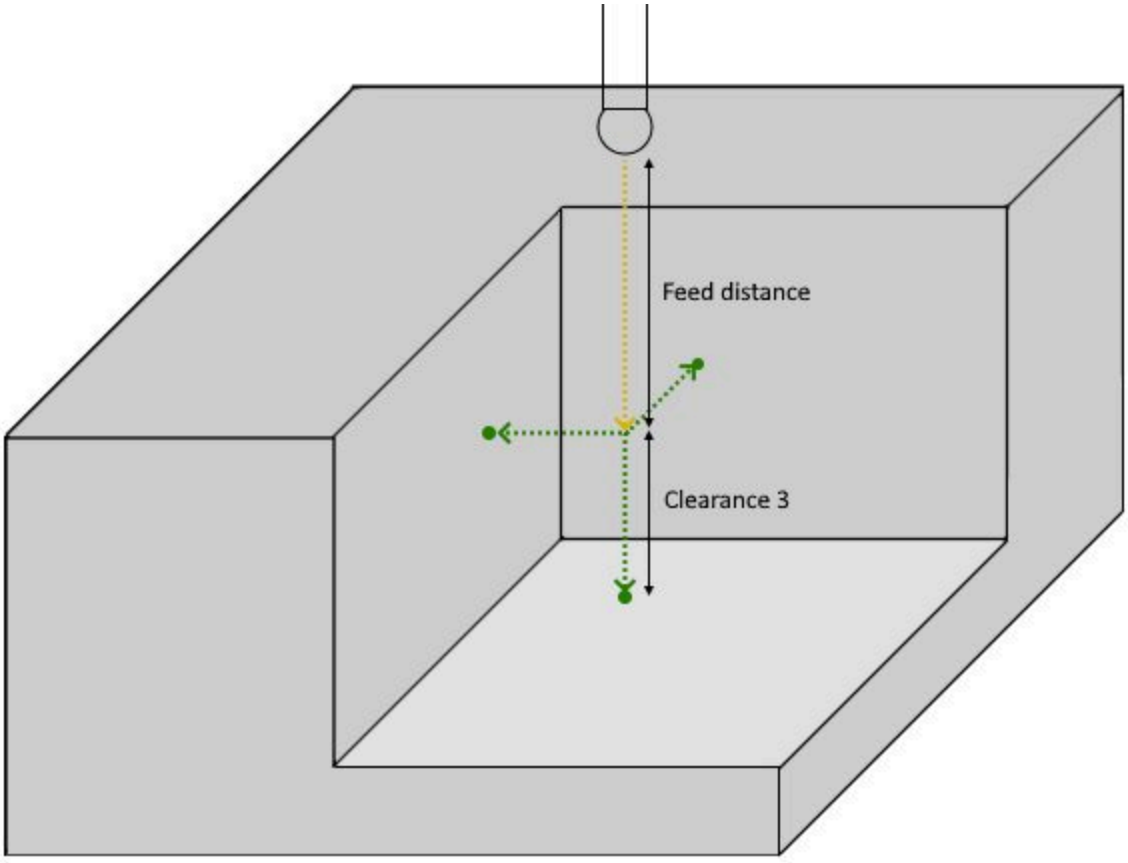
CLDArray	Type	Description
CmdPrm.Flt[-210]	Double	Original target vector of touch point on second wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Original target vector of touch point on second wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-212]	Double	Original touch point on top wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-213]	Double	Original touch point on top wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Original touch point on top wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Original target vector of touch point on top wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Original target vector of touch point on top wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-217]	Double	Original target vector of touch point on top wall value along Z-axis specified in the Job Assignment

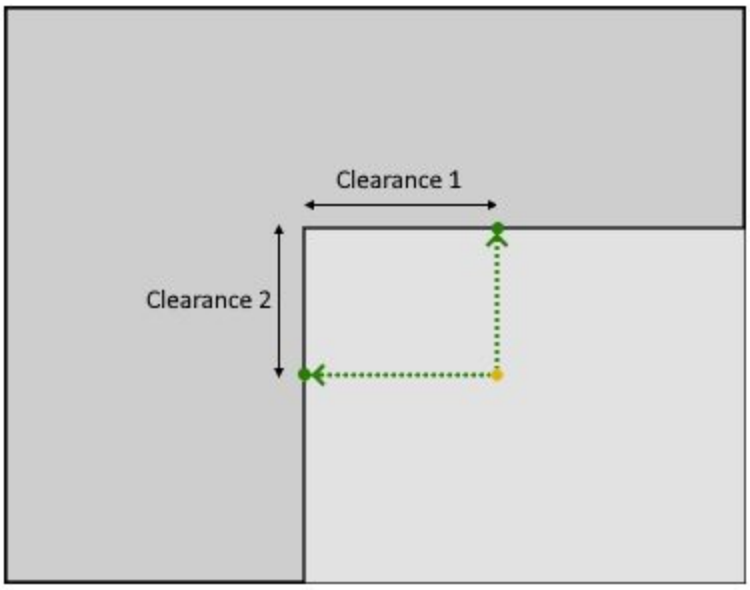
See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)

- CLData access model

Triple wall internal corner probing parameters





Triple wall internal corner probing consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting point of the measurement cycle that is the intersection of three touch points at a distance specified in `Clearance 1`, `Clearance 2`, `Clearance 3`, respectively, along their vectors. Moving at "Approach feed";

2. The tool approaches touch point of the first side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
3. The tool approaches touch point of the second side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
4. The tool approaches touch point of the third side (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Triple wall internal corner probing value = 15
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-56]	Double	Clearance 1, approach distance to touch point on first side
CmdPrm.Flt[-57]	Double	Clearance 2, approach distance to touch point on second side
CmdPrm.Flt[-58]	Double	Clearance 3, approach distance to touch point on third side
CmdPrm.Flt[-100]	Double	Touch point on first wall value along X-axis
CmdPrm.Flt[-101]	Double	T ouch point on first wall value along Y-axis
CmdPrm.Flt[-102]	Double	T ouch point on first wall value along Z-axis

CLDArray	Type	Description
CmdPrm.Flt[-103]	Double	T arget vector of touch point on first wall value along X-axis
CmdPrm.Flt[-104]	Double	T arget vector of touch point on first wall value along Y-axis
CmdPrm.Flt[-105]	Double	T arget vector of touch point on first wall value along Z-axis
CmdPrm.Flt[-106]	Double	T ouch point on second wall value along X-axis
CmdPrm.Flt[-107]	Double	T ouch point on second wall value along Y-axis
CmdPrm.Flt[-108]	Double	T ouch point on second wall value along Z-axis
CmdPrm.Flt[-109]	Double	T arget vector of touch point on second wall value along X-axis
CmdPrm.Flt[-110]	Double	T arget vector of touch point on second wall value along Y-axis
CmdPrm.Flt[-111]	Double	T arget vector of touch point on second wall value along Z-axis
CmdPrm.Flt[-112]	Double	T ouch point on third wall value along X-axis
CmdPrm.Flt[-113]	Double	T ouch point on third wall value along Y-axis
CmdPrm.Flt[-114]	Double	T ouch point on third wall value along Z-axis
CmdPrm.Flt[-115]	Double	T arget vector of touch point on third wall value along X-axis
CmdPrm.Flt[-116]	Double	T arget vector of touch point on third wall value along Y-axis

CLDArray	Type	Description
CmdPrm.Flt[-117]	Double	T arget vector of touch point on third wall value along Z-axis
CmdPrm.Flt[-200]	Double	Original touch point on first wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	Original touch point on first wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	Original touch point on first wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Original target vector of touch point on first wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	Original target vector of touch point on first wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Original target vector of touch point on first wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Original touch point on second wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Original touch point on second wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Original touch point on second wall value along Z-axis specified in the Job Assignment

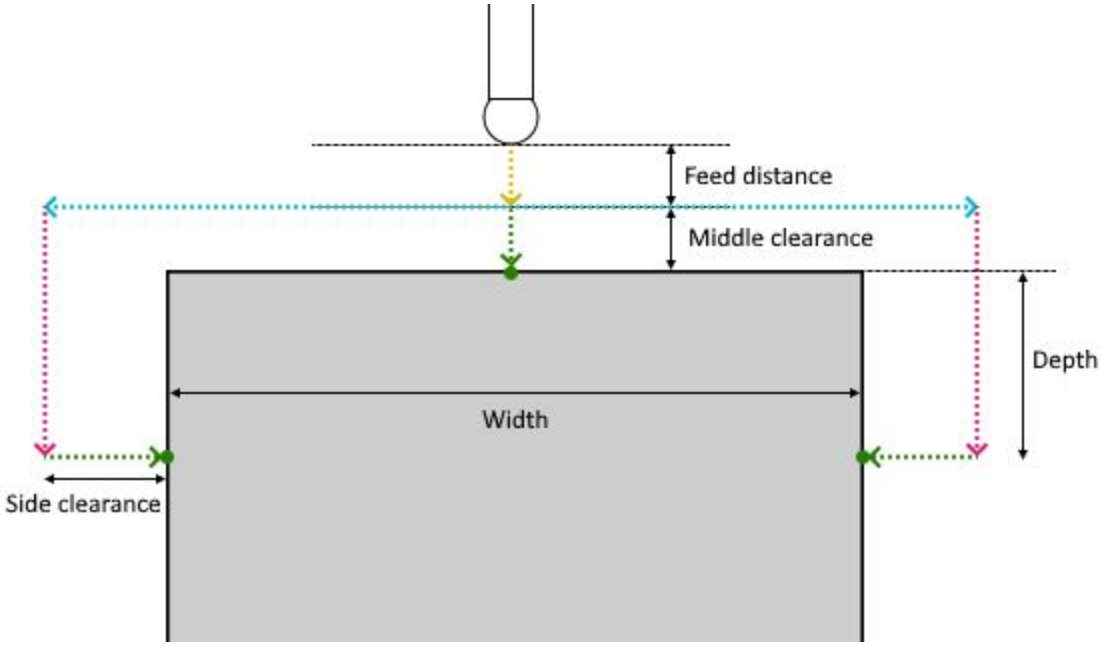
CLDArray	Type	Description
CmdPrm.Flt[-209]	Double	Original target vector of touch point on second wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Original target vector of touch point on second wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Original target vector of touch point on second wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-212]	Double	Original touch point on third wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-213]	Double	Original touch point on third wall value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Original touch point on third wall value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Original target vector of touch point on third wall value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Original target vector of touch point on third wall value along Y-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-217]	Double	Original target vector of touch point on third wall value along Z-axis specified in the Job Assignment

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Web probing (three points) parameters



Web probing (three points) consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting position of the cycle. Moving at "Approach feed";
2. The tool approaches a point remote from the left touch point along its `Target vector` at the `Side clearance` distance but elevated at the `Top clearance` value + `Depth` value. Moving at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. Moving at "Long link feed";
4. The tool approaches the left touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. Moving at "Long link feed";
6. The tool returns to the starting position;
7. Steps 2-6 are repeated for the right touch point;
8. The tool approaches the middle touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
9. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Web probing (three points) value = 10

CLDArray	Type	Description
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor " property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from middle side to center of left and right point
CmdPrm.Flt[-53]	Double	Width, distance from left point to right point
CmdPrm.Flt[-56]	Double	Side clearance, approach distance to right/left touch point
CmdPrm.Flt[-57]	Double	Middle clearance, approach distance to middle touch point
CmdPrm.Flt[-100]	Double	Left touch point value along X-axis
CmdPrm.Flt[-101]	Double	Left touch point value along Y-axis
CmdPrm.Flt[-102]	Double	Left touch point value along Z-axis
CmdPrm.Flt[-103]	Double	Left target vector value along X-axis
CmdPrm.Flt[-104]	Double	Left target vector value along Y-axis
CmdPrm.Flt[-105]	Double	Left target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Right touch point value along X-axis

CLDArray	Type	Description
CmdPrm.Flt[-107]	Double	Right touch point value along Y-axis
CmdPrm.Flt[-108]	Double	Right touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Right target vector value along X-axis
CmdPrm.Flt[-110]	Double	Right target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Right target vector value along Z-axis
CmdPrm.Flt[-112]	Double	Middle touch point value along X-axis
CmdPrm.Flt[-113]	Double	Middle touch point value along Y-axis
CmdPrm.Flt[-114]	Double	Middle touch point value along Z-axis
CmdPrm.Flt[-115]	Double	Middle target vector value along X-axis
CmdPrm.Flt[-116]	Double	Middle target vector value along Y-axis
CmdPrm.Flt[-117]	Double	Middle target vector value along Z-axis
CmdPrm.Flt[-200]	Double	Left original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	Left original touch point value along Y-axis specified in the Job Assignment

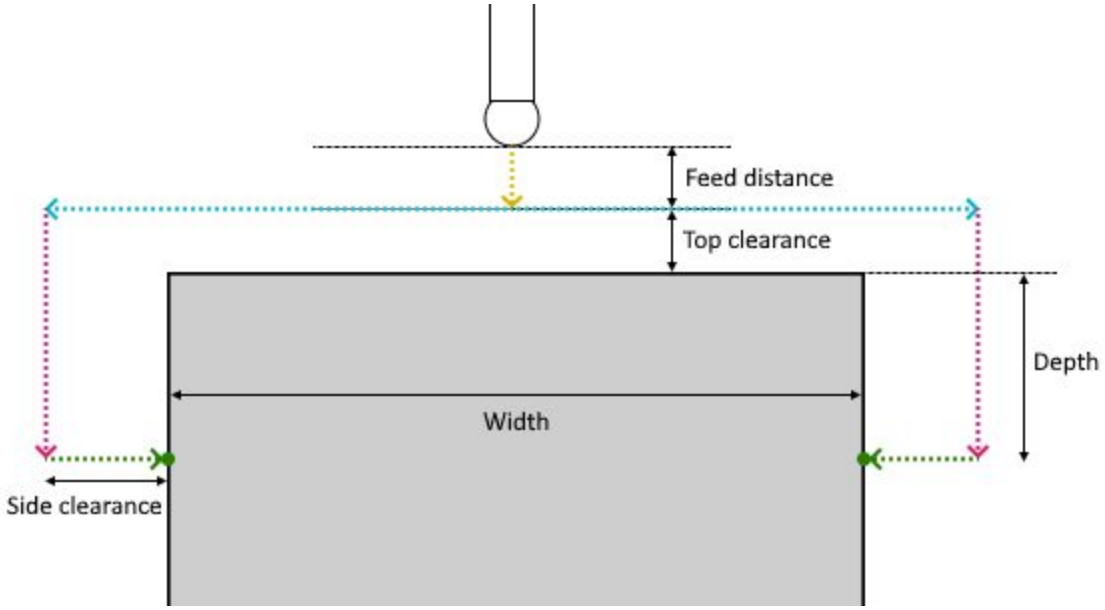
CLDArray	Type	Description
CmdPrm.Flt[-202]	Double	Left original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	Left original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	Left original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	Left original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Right original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-207]	Double	Right original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Right original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Right original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Right original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Right original target vector value along Z-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-212]	Double	Middle original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-213]	Double	Middle original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-214]	Double	Middle original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-215]	Double	Middle original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-216]	Double	Middle original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-217]	Double	Middle original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

Web probing parameters



Web probing consist of the following steps:

1. The tool moves the distance specified in the `Feed distance` and approaches the starting position of the cycle. Moving at "Approach feed";
2. The tool approaches a point remote from the first touch point along its `Target vector` at the `Side clearance` distance but elevated at the `Top clearance` value + `Depth` value. Moving at "Long link feed";
3. The tool is lowered by a distance equal to `Top clearance` value + `Depth` value. Moving at "Long link feed";
4. The tool approaches the first touch point (moving at "Work feed") and then returns (moving at "Long link feed") to previous position;
5. The tool is lifted up by a distance equal to `Top clearance` value + `Depth` value. Moving at "Long link feed";
6. The tool returns to the starting position;
7. Steps 2-6 are repeated for the second touch point;
8. The tool is lifted up by a distance equal to `Feed distance`.

Parameters

CLDArray	Type	Description
CmdPrm.Int[-1]	Integer	Probing cycle type: Web probing value = 9
CmdPrm.Int[-2]	Integer	SubCode of cycle specified in " SubCode for postprocessor "

CLDArray	Type	Description
		property on the `Job Assignment` tab
CmdPrm.Flt[-50]	Double	Feed distance, distance to start position of cycle
CmdPrm.Flt[-51]	Double	Depth, distance from top side to center of touch points
CmdPrm.Flt[-53]	Double	Width, distance from Touch point 1 to Touch point 2
CmdPrm.Flt[-55]	Double	Top clearance, distance from top side
CmdPrm.Flt[-56]	Double	Side clearance, approach distance to touch point 1/ touch point 2
CmdPrm.Flt[-100]	Double	First touch point value along X-axis
CmdPrm.Flt[-101]	Double	First touch point value along Y-axis
CmdPrm.Flt[-102]	Double	First touch point value along Z-axis
CmdPrm.Flt[-103]	Double	First target vector value along X-axis
CmdPrm.Flt[-104]	Double	First target vector value along Y-axis
CmdPrm.Flt[-105]	Double	First target vector value along Z-axis
CmdPrm.Flt[-106]	Double	Second touch point value along X-axis
CmdPrm.Flt[-107]	Double	Second touch point value along Y-axis

CLDArray	Type	Description
CmdPrm.Flt[-108]	Double	Second touch point value along Z-axis
CmdPrm.Flt[-109]	Double	Second target vector value along X-axis
CmdPrm.Flt[-110]	Double	Second target vector value along Y-axis
CmdPrm.Flt[-111]	Double	Second target vector value along Z-axis
CmdPrm.Flt[-200]	Double	First original touch point value along X-axis specified in the Job Assignment
CmdPrm.Flt[-201]	Double	First original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-202]	Double	First original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-203]	Double	First original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-204]	Double	First original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-205]	Double	First original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-206]	Double	Second original touch point value along X-axis specified in the Job Assignment

CLDArray	Type	Description
CmdPrm.Flt[-207]	Double	Second original touch point value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-208]	Double	Second original touch point value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-209]	Double	Second original target vector value along X-axis specified in the Job Assignment
CmdPrm.Flt[-210]	Double	Second original target vector value along Y-axis specified in the Job Assignment
CmdPrm.Flt[-211]	Double	Second original target vector value along Z-axis specified in the Job Assignment
CmdPrm.Flt[-66]	Integer	Compensate dimensions, retrurn values: 0 - Off, 1 - On

See also

- Probing cycle (WProbing)
- Extended cycle (EXTCYCLE)
- CLData access model

FEDRAT - Feedrate

Command caption

How the command appears in the CLData command list:

```
FEDRAT NM nm, K k, MMPM(315) | MMPR(316)
```

FEDRAT defines the value and the type of tool feed. The feed value is stored until the next FEDRAT command or RAPID command.

CAM system uses FEDRAT to pass the machining condition code.

Access from sppx

Handler: `program Fedrat`

Parameter	CLD array	CLD name	Description
nm	CLD[1]	CLD.NM	Feed value.
k	CLD[2]	CLD.K	Feed type (work feed, approach feed, retract feed, transition feed, cut-in feed, etc.). 0 - working, 2 - first pass feed, 4 - engage feed, 8 - retract feed, 16 - plunge feed, 32 - finish pass feed 64 - transition to the next pass, 128 - return feed, 256 - approach feed. For EDM operation CAM system uses this parameter to pass machining condition code.

Parameter	CLD array	CLD name	Description
MMPM(315) MMPR(316)	CLD[3]	CLD.MMPM	Feedrate units: MMPM(315) – mm per minute, MMPR(316) – mm per revolution

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Fedrat
  FEED_ = cld[1]   ! filling FEED_ rgister
  if INTERP_ = 0 then INTERP_ = 1   ! G1
  if cld[3]=315 then GFeed = 94
                else GFeed = 95
end
```

Access from .NET

Handlers:

```
public override void OnMoveVelocity(ICLDMoveVelocityCommand cmd, CLDArray cld)   //
shared velocity handler (FEDRAT and RAPID)
public override void OnFeedrate(ICLDFeedrateCommand cmd, CLDArray cld)   //
working feed
```

Inheritance: `ICLDFeedrateCommand : ICLDMoveVelocityCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsRapid`	`bool`	Contains "true" if the command is a RAPID command, otherwise it is "false".
`cmd.FeedKind`	`CLDFeedKind`	The kind of a feed (working, rapid, engage, return, finish etc.). Possible values:

Member	Type	Description
		`Working` (0) - The feed kind for base working movements. `Rapid` (1) - The feed kind of rapid movements. `First` (2) - The feed kind for the working movements on a first (roughing) pass. `Engage` (4) - The feed kind for the movements entering to the first machining (cutting) point. `Retract` (8) - The feed kind for the movements leaving the last machining (cutting) point. `Plunge` (16) - The feed kind for the movements that plunge vertically into the machining area. `Finish` (32) - The feed kind for the working movements on a last (finishing) pass. `Next` (64) - The feed kind for the short link movements. `Return` (128) - The feed kind for the movements which take the tool away from the machining site. `Approach` (256) - The feed kind for the movements which bring the tool closer to the machining site. `Rapid5D` (512) - The feed kind value reserved for an internal use. `TransitionOnSafe` (1024) - The feed kind for the movements performed in a safe area. `ApproachFromSafe` (2048) - The feed kind for the movements from the safe area to the

Member	Type	Description
		machining side. `ReturnToSafe` (4096) - The feed kind for the movements from the machining side to the safe area. `LongNext` (8192) - The feed kind for the long link movements. `ThreadPitch` (16384) - The feed kind for the threading movements. `Overload` (1073741824) - The feed kind value reserved for an internal use.
`cmd.FeedCode`	`int`	Numerical representation of the "FeedKind" flag, that defines the kind of a feed (working, rapid, engage, return, finish etc.). For a wire EDM case it contains the numerical code of a row in a machining conditions table.
`cmd.FeedUnits`	`CLDFeedUnits`	Units of the feed (mm/min, mm/rev etc.) Possible values: `MPM` (0) - Current linear units (mm or inch) per minute. `MPR` (1) - Current linear units (mm or inch) per revolution. `ConditionCode` (2) - Numerical code of a row in a machining conditions table (for a wire EDM).
`cmd.FeedValue`	`double`	The value of the feed.

Example - `OnMoveVelocity` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnMoveVelocity(ICLDMoveVelocityCommand cmd, CLDArray cld)
{
```

```
    if (cmd.IsRapid) {
        nc.GInterp.v = 0;
    } else {
        if (nc.GInterp == 0)
            nc.GInterp.v = 1;
        nc.F.v = cmd.FeedValue;
        if (cmd.FeedUnits==CLDFeedUnits.MPR)
            nc.GFeed.v = 95;
        else
            nc.GFeed.v = 94;
        if (IsFirstFeed)
        {
            nc.F.Show();
            nc.GFeed.Show();
            IsFirstFeed = false;
        }
    }
}
```

See also

- CLData access model

Filter - Output filter (global handler)

> Not a CLData command. `Filter` is a global handler called on every NC-program line, whatever CLData command (or any other code) produced it. It is listed here so that all postprocessor handlers can be found in one place and in a common style.

Access from sppx

All postprocessors include the predefined subprogram Filter, which can be found in the subprograms list. Filter is called immediately after the NC-program block was formed and before it is added to the NC-program text. Thus, this subprogram allows the user to edit every NC-program block at the moment of its output. For example, it is possible to replace symbols or add marks at the end or at the beginning of a line.

By default the system creates the following subprogram prototype for the Filter:

```
sub Filter(S: string)
subend
```

The formed NC-block is passed through the S parameter. All changes to that string will be put into the NC-program text.

Example:

```
sub Filter(S: String)
  replace(S,"R1","L") ! Replace "R1" with "L"
  replace(S,"R-1","R") ! Replace "R-1" with "R"
  replace(S,"F0","FMAX") ! Replace "F0" with "FMAX"
subend
```

Executing this code for the block "17 L Z-10 R0 F0" will result with the line of NC-program "17 L Z-10 R0 FMAX".

Access from .NET

The .NET counterpart is the `OnFilterString` handler:

```
public override void OnFilterString(ref string s, TNCFile ncFile, INCLabel label)
```

The line is passed by reference in `s` - modify it in place. `ncFile` is the target NC file, `label` the current label.

Example (from the Goodway SW postprocessor):

```
public override void OnFilterString(ref string s, TNCFile ncFile, INCLabel label)
{
    SynchPoints.CheckForOutput(s);
    if ((ncFile is NCFile ncx) && (ncx.NeedSlash) && (!String.IsNullOrEmpty(s)))
        s= "/"+s;
}
```

See also

- CLData access model

FINI - Ending record

Command caption

How the command appears in the CLData command list:

```
FINI
```

FINI command is the last command in the list of technology commands of the project. Use FINI to form NC-program conclusion blocks.

Access from sppx

Handler: `program Fini`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Fini
  OutBlock                          ! output
  OutPut "M30"
  OutPut "%"
  OutPut ""
  NCSub.Output

  s$ = "G53Z0."
  CrLf$ = Chr(13) + Chr(10)
  if (HasAxisA<>0) or (HasAxisB<>0) or (HasAxisC<>0) then begin
    s$ = s$ + CrLf$ + "G53"
    if (HasAxisA<>0) then
      s$ = s$ + "A0."
    if (HasAxisB<>0) then
      s$ = s$ + "B0."
    if (HasAxisC<>0) then
      s$ = s$ + "C0."
  end
  ReplNCStr("<StartGoHome>", s$)
end
```

Access from .NET

Handlers:

```
public override void OnFini(ICLDFiniCommand cmd, CLDArray cld)
public override void OnFinishProject(ICLDProject prj)   // project-level, raised
on FINI
```

Inheritance: `ICLDFiniCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Example - `OnFinishProject` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnFinishProject(ICLDProject prj)
{
    nc.Block.Out();
    nc.Output("M30");
}
```

See also

- CLData access model

FROM - Original point

Command caption

How the command appears in the CLData command list:

```
FROM COUNT 5, MACHINE, X 0, Y 0, Z 254, A 0, C 0, GEOM X 10, Y 0, Z 194, NX 0, NY 0,
NZ 1, NW 0, MSF 0, Time 0
```

FROM command defines original point coordinates. Usually FROM command is called only once at the beginning of machining, the first transition coordinates commonly equal the FROM command coordinates. If is necessary to avoid output of original point into the NC-program use the FROM command to initialize register values without printing them out into the NC-program code. This prevents the output of movement commands if the travel coordinates equal those of the FROM command.

Access from sppx

Handler: `program From`

Parameter	CLD array	CLD name	Description
X Y Z	CLD[1] CLD[2] CLD[3]	CLD.X CLD.Y CLD.Z	Original tooltip point spatial coordinates, relative to the active workpiece CS
NX NY NZ NW	CLD[4] CLD[5] CLD[6] CLD[7]	CLD.NX CLD.NY CLD.NZ CLD.NW	Original tool frame orientation, relative to the active workpiece CS. The way of orientation definition depend on setting inside Machine schema. It can be Euler angles in different conventions, Tool normal vector, quaternion or rotated vector.
MSF	CLD[8]	CLD.MSF	Machine state flags (flips). Actual for the

Parameter	CLD array	CLD name	Description
			robots and machines that can have multiple configurations of physical axes for same spatial position. See description in the next table.
Time	CLD[9]	CLD.Time	The time of movement.
AxesCount	CLD[10]	CLD.AxesCount	Number of physical machine coordinates affected in this command.
r1	CLD[11]		Index of axis named Axis1Pos in the coordinates list.
Val1	CLD[12]		Value of the machine coordinate Axis1Pos.
r2	CLD[13]		Index of axis named Axis2Pos in the coordinates list.
Val2	CLD[14]		Value of the machine coordinate Axis2Pos.
...	...		...
rN	CLD[2*N+9]		Index of axis named AxisNPos in the coordinates list.
ValN	CLD[2*N+10]		Value of the machine coordinate AxisNPos.

Parameters available through the cmd operator

TCLDFrom: ComplexType	Original point coordinates command
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - Separate element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["Pos5D.X"] - X cartesian coordinate of the tool tip position.

TCLDFrom: ComplexType	Original point coordinates command
Y: Double	cmd.Flt["Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["Pos5D.Z"] - Z cartesian coordinate of the tool tip position.
NX: Double	cmd.Flt["Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["Pos5D.NY"] - depending on the system configuration may be the NY component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["MachineStateFlags"].Bit[i], i=(0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of these solutions. For a standard six-axis

TCLDFrom: ComplexType	Original point coordinates command
	articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.
Time: Double	The time of the movement in minutes.

The list of coordinates passed through CLData of FROM command is defined by the CAM system machine scheme.

Use the GMA array to process the FROM command. The properties of the GMA array are automatically assigned precalculated values of coordinate movement parameters. The command parameters are also available through the cmd operator.

Example (from the Heidenhain (iTNC530)_Mill postprocessor):

```
program From
  !X = CLD[1]; X@ = X; XT_ = X
  !Y = CLD[2]; Y@ = Y; YT_ = Y
  !Z = CLD[3]; Z@ = Z; ZT_ = Z
  !A = 0; A@ = A; AT_ = A
  !B = 0; B@ = B; BT_ = B
  !C = 0; C@ = C; CT_ = C
End
```

Access from .NET

Handlers:

```
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
group handler - before any motion command
public override void OnFrom(ICLDFromCommand cmd, CLDArray cld)
```

```
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)   // group
handler - after any motion command
```

Inheritance: `ICLDFromCommand : ICLDMultiMotionCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.EN`	`TInpRotation`	End normal or orientation of the movement in cartesian coordinate system. It can be normal vector, euler angles, quaternion etc. depend on the current machine schema settings.
`cmd.MSFlags`	`BitArray`	Machine state flags or robot flips which can help to define (together with cartesian coordinates) the specific state of the mechanism in case of redundancy. Which bit is responsible for which flag depends on the type and configuration of the equipment.
`cmd.Time`	`double`	Duration of the movement in minutes.
`cmd.AxesCount`	`int`	The count of axes that the movement contains inside the Axis[] list.
`cmd.Axis`	`ICLDMultiMotionAxesIndexer`	The list of the movement's axes [0..(AxesCount-1)].
`cmd.Axes`	`IEnumerable`	Axes' enumerator object for the movement to be possible to use

Member	Type	Description
		covenient foreach syntax.
`cmd.HasAxis(...)`	`bool`	Returns "true" if the movement contains axis with the speciefed textual ID. Otherwise returns "false".
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement

Example - `OnFrom` (from the Sinumerik (840D)_Mill_DN postprocessor):

```
public override void OnFrom(ICLDFromCommand cmd, CLDArray cld)
{
    nc.X.Hide(cld[1]); nc.Y.Hide(cld[2]); nc.Z.Hide(cld[3]);
    LastPnt.X = cld[1]; LastPnt.Y = cld[2]; LastPnt.Z = cld[3];
}
```

See also

- CLData access model

GOHOME - Return to original position

Command caption

How the command appears in the CLData command list:

```
GOHOME COUNT 2, MACHINE, X 34.725, Y 32, GEOM X 34.725, Y 32, Z 194, NX 0, NY 0, NZ
1, NW 0, MSF 0, Time 0.003
```

GOHOME command is used to move the tool to the machine zero. Commonly GOHOME command is translated into G28 code. Machine zero is defined by the CNC, postprocessor only receives the command code.

Access from sppx

Handler: `program GoHome`

Parameter	CLD array	Description
X Y Z	CLD[1] CLD[2] CLD[3]	Tooltip spatial coordinates, relative to the active workpiece CS
NX NY NZ NW	CLD[4] CLD[5] CLD[6] CLD[7]	Tool frame orientation, relative to the active workpiece CS. The way of orientation definition depend on setting inside Machine schema. It can be Euler angles in different conventions, Tool normal vector, quaternion or rotated vector.
MSF	CLD[8]	Machine state flags (flips). Actual for the robots and machines that can have multiple configurations of physical axes for same spatial position. See description in the next table.
Time	CLD[9]	The time of movement.

Parameter	CLD array	Description
AxesCount	CLD[10]	Number of physical machine coordinates affected in this command.
r1	CLD[11]	Index of axis named Axis1Pos in the coordinates list.
Val1	CLD[12]	Value of the machine coordinate Axis1Pos.
r2	CLD[13]	Index of axis named Axis2Pos in the coordinates list.
Val2	CLD[14]	Value of the machine coordinate Axis2Pos.
...	...	...
rN	CLD[2*N+9]	Index of axis named AxisNPos in the coordinates list.
ValN	CLD[2*N+10]	Value of the machine coordinate AxisNPos.

Parameters available through the cmd operator

TCLDGoHome: ComplexType	Return machine to home position command
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - Separate element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.

TCLDGoHome: ComplexType	Return machine to home position command
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["Pos5D.X"] - X cartesian coordinate of the tool tip position.
Y: Double	cmd.Flt["Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["Pos5D.Z"] - Z cartesian coordinate of the tool tip position.
NX: Double	cmd.Flt["Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["Pos5D.NY"] - depending on the system configuration may be the NY component of the

TCLDGoHome: ComplexType	Return machine to home position command
	tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["MachineStateFlags"].Bit[i], i=(0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of these solutions. For a standard six-axis articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.

TCLDGoHome: ComplexType	Return machine to home position command
Time: Double	The time of the movement in minutes.

The list of coordinates passed through CLData of GOHOME command is defined by the CAM system machine scheme.

Use the GMA array to process the GOHOME command. The properties of the GMA array are automatically assigned precalculated values of coordinate movement parameters. The command parameters are also available through the cmd operator.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program GoHome
  if CycleOn=1 then begin
    CycleOn = 0
    Cycle = 80
  end
  !OutBlock                ! Output in block

  if Cmd.Ptr["Axes(AxisXPos)"]<>0 then begin
    X = 0; X@ = MaxReal
  end
  if Cmd.Ptr["Axes(AxisYPos)"]<>0 then begin
    Y = 0; Y@ = MaxReal
  end
  if Cmd.Ptr["Axes(AxisZPos)"]<>0 then begin
    Z = 0; Z@ = MaxReal
  end
  if Cmd.Ptr["Axes(AxisAPos)"]<>0 then begin
    A = 0; A@ = MaxReal
    HasAxisA = 1
  end
  if Cmd.Ptr["Axes(AxisBPos)"]<>0 then begin
    B = 0; B@ = MaxReal
    HasAxisB = 1
  end
  if Cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
    C = 0; C@ = MaxReal
    HasAxisC = 1
  end
  if (X<>X@) or (Y<>Y@) or (Z<>Z@) or
     (A<>A@) or (B<>B@) or (C<>C@)
  then begin
    GFunc = 28; GFunc@ = MaxReal
```

```
    GAbsInc = 91
    OutBlock
    if NextCode<>CodeOfCmd("GoHome") then begin
      GAbsInc = 90
      OutBlock
    end
  end
end
```

Access from .NET

Handlers:

```
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
group handler - before any motion command
public override void OnGoHome(ICLDGoHomeCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)   // group
handler - after any motion command
```

Inheritance: `ICLDGoHomeCommand : ICLDMultiMotionCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.EN`	`TInpRotation`	End normal or orientation of the movement in cartesian coordinate system. It can be normal vector, euler angles, quaternion etc. depend on the current machine schema settings.
`cmd.MSFlags`	`BitArray`	Machine state flags or robot flips which can help to define (together with cartesian coordinates) the specific state of the mechanism in case of redundancy. Which bit is

Member	Type	Description
		responsible for which flag depends on the type and configuration of the equipment.
`cmd.Time`	`double`	Duration of the movement in minutes.
`cmd.AxesCount`	`int`	The count of axes that the movement contains inside the Axis[] list.
`cmd.Axis`	`ICLDMultiMotionAxesIndexer`	The list of the movement's axes [0..(AxesCount-1)].
`cmd.Axes`	`IEnumerable`	Axes' enumerator object for the movement to be possible to use covenient foreach syntax.
`cmd.HasAxis(...)`	`bool`	Returns "true" if the movement contains axis with the speciefed textual ID. Otherwise returns "false".
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement

Example - `OnGoHome` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnGoHome(ICLDGoHomeCommand cmd, CLDArray cld)
{
    foreach(CLDMultiMotionAxis ax in cmd.Axes) {
        if (ax.IsX) {
            nc.U.Show(0);
        } else if (ax.IsY) {
            nc.V.Show(0);
        } else if (ax.IsZ) {
            nc.W.Show(0);
        }
    }
    if (nc.U.Changed || nc.V.Changed || nc.W.Changed) {
        nc.GHome.Show(28);
        nc.Block.Out();
```

```
    }
}
```

See also

- CLData access model

GOTO - Linear transition

Command caption

How the command appears in the CLData command list:

```
GOTO.abs X x, Y y, Z z
```

CAM system passes linear transition data through ABSMOV (`GOTO.abs`) command. The parameters of the command are the new absolute coordinates along the X, Y and Z axes.

Access from sppx

Handler: `program AbsMov`

Parameter	CLD array	CLD name	Description
x y z	CLD[1] CLD[2] CLD[3]	CLD.X CLD.Y CLD.Z	New tool coordinates (absolute)

Example (from the Fanuc (30i)_Mill postprocessor):

```
program AbsMov
  if (INTERP_ > 1) and (INTERP_ <> 33) then
    INTERP_ = 1    ! G1
  X = cld[1]                          ! X,Y,Z in absolutes
  Y = cld[2]
  if IsRotaryInterpolation = 2 then begin
    Y = -Y/CylR/Pi*180
  end
  Z = cld[3]
  if (Z<>Z@) and (NeedKorDl<>0) then begin
    KorDl = NeedKorDl; KorDl@ = MaxReal
    H@ = MaxReal
    NeedKorDl = 0
  end
  if (CycleOn=0) and ((X<>X@) or (Y<>Y@) or (Z<>Z@)) then begin
    if KorEcv<>KorEcv@ then begin
      if Interp_<>Interp_@ then begin
        X@ = X
        Y@ = Y
        Z@ = Z
```

```
        KorEcv@ = KorEcv
        D@ = D
        OutBlock
        KorEcv@ = MaxReal
        D@ = MaxReal
      end
      X@ = MaxReal !XT_
      Y@ = MaxReal !YT_
      Z@ = MaxReal !ZT_
    end
    if IsMultiaxisInterpolation = 1 then begin
      X@ = MaxReal;
      Y@ = MaxReal;
      Z@ = MaxReal;
    end;
    OutBlock
  end
  XT_ = X;  YT_ = Y;  ZT_ = Z        ! save current coordinates
end
```

Access from .NET

Handlers:

```
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
group handler - before any motion command
public override void OnGoto(ICLDGotoCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)   // group
handler - after any motion command
```

Inheritance: `ICLDGotoCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement

Example - `OnGoto` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnGoto(ICLDGotoCommand cmd, CLDArray cld)
{
    if (nc.GInterp > 1)
        nc.GInterp.v = 1;
    nc.X.v = cmd.EP.X;
    nc.Y.v = cmd.EP.Y;
    nc.Z.v = cmd.EP.Z;

    if (!cycleIsOn) {
        if (nc.Z.Changed && nc.GLCompens.ValuesDiffer) {
            nc.Block.Hide(nc.X, nc.Y);
            nc.Block.Show(nc.GLCompens, nc.HLCompens, nc.Z);
            nc.Block.Out();
            nc.Block.UpdateState(nc.X, nc.Y);
        }
        // Uncomment if you want mandatory XYZABC in TCPM mode
        // if (nc.GLCompens == 43.4) {
        //     nc.Block.Show(nc.X, nc.Y, nc.Z);
        //     if (CLDProject.Machine.HasAAxis)
        //         nc.A.Show();
        //     if (CLDProject.Machine.HasBAxis)
        //         nc.B.Show();
        //     if (CLDProject.Machine.HasCAxis)
        //         nc.C.Show();
        // }
        if(nc.X.Changed || nc.Y.Changed || nc.Z.Changed )
            nc.Block.Out();
    }
    nc.LastP = cmd.EP;
}
```

See also

- CLData access model

HEAD - Spindle head number

> Deprecated. This command is obsolete - the current version of the CAM system does not generate it automatically. It is documented here for postprocessors that still handle legacy CLData.

Command caption

How the command appears in the CLData command list:

```
HEAD BOTH(83)|N n
```

Use HEAD command to select active spindle head. The head number is passed as parameter.

Access from sppx

Handler: `program Head`

Parameter	CLD array	CLD name	Description
BOTH(83) n	CLD[1]	CLD.N	83 (BOTH) – simultaneous operation of both heads, n – head number.

See also

- CLData access model

INSERT - Insertion

Command caption

How the command appears in the CLData command list:

```
INSERT "..."
```

INSERT command is used to pass textual information directly into NC-program. The text is put into CLDATA$ variable.

Access from sppx

Handler: `program Insert`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Insert
  OutBlock
  FormBlock
  Output OutStr$ + CLData$
end
```

Access from .NET

Handler:

```
public override void OnInsert(ICLDInsertCommand cmd, CLDArray cld)
```

Inheritance: `ICLDInsertCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.Text`	`string`	The text to output to the file.

Example - `OnInsert` (from the Heidenhain_Mill_iTNC530_DN postprocessor):

```
public override void OnInsert(ICLDInsertCommand cmd, CLDArray cld)
{
    nc.WriteLine(cmd.Text);
}
```

See also

- CLData access model

INTERPOLATION - Interpolation mode

Command caption

How the command appears in the CLData command list:

```
INTERPOLATION ON(71) | OFF(72), CARTESIAN(9020) | POLAR(9021) |
CYLINDRICAL(9022) | MULTIAXIS(9023), P1 p1, P2 p2, P3 p3
```

INTERPOLATION command activates or deactivates interpolation modes like cylindrical or polar interpolation.

Interpolation mode specified by the `CLD[2]` (`CLD.Mode`) parameter is activated if `CLD[1]` (`CLD.SubCmd`) parameter is `ON(71)`, otherwise the specified interpolation is deactivated.

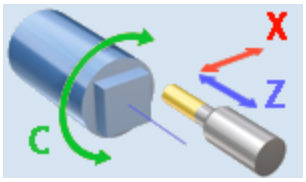
Parameters `CLD[3]` (`CLD.P1`), `CLD[4]` (`CLD.P2`), `CLD[5]` (`CLD.P3`) specify additional parameters of interpolation modes.

Access from sppx

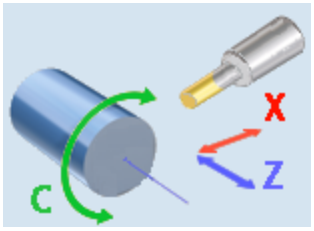
Handler: `program Interpolation`

Parameter	CLD array	CLD name	Description
ON(71) OFF(72)	CLD[1]	CLD.SubCmd	ON(71) – activate, OFF(72) – deactivate.
CARTESIAN(9020) POLAR(9021) CYLINDRICAL(9022) MULTIAXIS(9023)	CLD[2]	CLD.Mode	CARTESIAN(9020) – cartesian interpolation; POLAR(9021) – polar interpolation; CYLINDRICAL(9022) – cylindrical interpolation; MULTIAXIS(9023) – multiaxis interpolation
P1	CLD[3]	CLD.P1	Specifies cylinder radius for the cylindrical interpolation mode
P2	CLD[4]	CLD.P2	Reserved
P3	CLD[5]	CLD.P3	Reserved

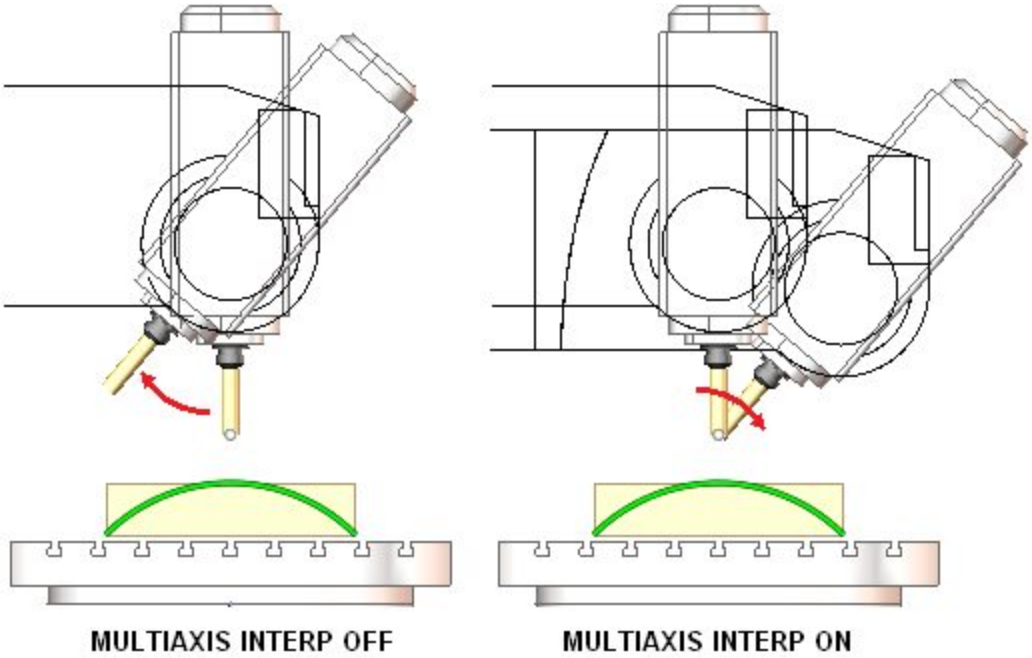
The Polar interpolation mode is usually used on turn-milling machines without the Y axis in order to machine parts from the side. In this mode the tool movement in Y direction is replaced by the simultaneous rotation of the workpiece around the rotary axis (C) and the tool movement along the radial axis (X). Depending on the control used the Y coordinate in the NC code may be replaced with the C coordinate.



The Cylindrical interpolation mode is used to machine cylindrical pockets on cylindrical parts. The NC code contains a 2d contour of a pocket in the XY coordinates where the X coordinate is the coordinate along the cylinder axis, while the Y coordinate is equal to the Cylinder Radius multiplied by the rotation Angle of the workpiece around the rotary axis. Depending on the control used the Y coordinate in the NC code may be replaced with the C coordinate.



The MULTIAXIS interpolation mode is used for simultaneous five axis milling. In this mode the XYZ coordinates of a NC block are the coordinates of the tool center point relative to the workpiece coordinate system rotating together with the rotary table. Opposed to the standard Cartesian interpolation mode in the MULTIAXIS interpolation mode if a NC block consists only of the commands positioning rotary axes, the position of the tool tip to the workpiece remains the same. It means the tool is rotated around the tool center (see the figure below).



If you program a tilting table movement while multiaxis interpolation is active, the control rotates the coordinate system accordingly. If, for example, you rotate the C axis by 90° (through a positioning command or datum shift) and then program a movement in the X axis, the control executes the movement in the machine axis Y.

Different controls use different commands for enabling and disabling the Multiaxis interpolation mode. In Heidenhain the mode is known as the Tool center point management controlled by the TCPM, M128 and M129 functions. In Sinumeric the mode is turned on/off by the TRAORI and TRAFOOF commands.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Interpolation
  case cld[2] of
    9023: begin ! 5-axis
      if CLD.SubCmd=71 then begin ! On
        NeedKorDl = 43.4
        IsMultiaxisInterpolation = 1
      end else begin              ! Off
        KorDl = 49;! KorDl@ = MaxReal
        IsMultiaxisInterpolation = 0
      end
      OutBlock
    end
    9021: begin ! POLAR
      OutBlock
      if CLD.SubCmd=71 then begin ! On
        GInterpRot = 12.1
        YAddress = "A"
        IsRotaryInterpolation = 1
```

```
      end else begin              ! Off
        GInterpRot = 13.1
        YAddress = "Y"
        IsRotaryInterpolation = 0
      end
      GInterpRot@ = MaxReal
      OutBlock
    end
    9022: begin ! CYLINDRICAL
      OutBlock
      if CLD.SubCmd=71 then begin ! On
        GInterpRot = 07.1
        A = CLD.P1; A@ = MaxReal
        YAddress = "A"
        IsRotaryInterpolation = 2
        CylR = CLD.P1
        CirclesThroughRadius = 1
      end else begin              ! Off
        GInterpRot = 07.1
        A = 0; A@ = MaxReal
        YAddress = "Y"
        IsRotaryInterpolation = 0
        CirclesThroughRadius = 0
      end
      GInterpRot@ = MaxReal
      OutBlock
    end
  end
end
```

Access from .NET

Handlers:

```
public override void OnInterpolation(ICLDInterpolationCommand cmd, CLDArray cld)
public override void OnInterpPolar(ICLDInterpolationCommand cmd, CLDArray cld)
// POLAR(9021)
public override void OnInterpCylindrical(ICLDInterpolationCommand cmd, CLDArray cld)
// CYLINDRICAL(9022)
public override void OnInterp5x(ICLDInterpolationCommand cmd, CLDArray cld)
// MULTIAXIS(9023)
```

Inheritance: `ICLDInterpolationCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Returns "True" if it's interpolation switch ON command, otherwise it returns "False".
`cmd.IsOff`	`bool`	Returns "True" if it's interpolation switch OFF command, otherwise it returns "False".
`cmd.InterpMode`	`CLDInterpMode`	Defines the type of an interpolation (polar-G112, cylindrical-G107 or continuous multiaxis-G43.4, TCPM, TRAORI).
`cmd.InterpType`	`int`	Numerical code to specify the type of an interpolation (polar, cylindrical or continuous multiaxis).
`cmd.CylRadius`	`double`	Specifies the cylinder radius for the cylindrical interpolation mode.

Example - `OnInterpolation` (from the Sinumerik (840D)_Mill_DN postprocessor):

```
public override void OnInterpolation(ICLDInterpolationCommand cmd, CLDArray cld)
{
    if (cmd.InterpType == 9023){// MULTIAXIS interpolation
        if (cmd.IsOn){ // Switch on
            nc.Block.Out();
            nc.WriteLineWithBlockN($"TRAORI");
            nc.CoordSys.v0 = double.MaxValue; nc.Block.Out();
            nc.WriteLineWithBlockN($"ORIWKS");
            nc.WriteLineWithBlockN($"ORIAXES");
```

```
        }else{          // Switch off
            nc.Block.Out();
            nc.WriteLineWithBlockN($"TRAFOOF");
        }
    }else if (cmd.InterpType == 9021) {// Polar interpolation
        if (cmd.IsOn) { // Switch on
            nc.Block.Out();
            nc.WriteLine("TRANSMIT");
            Cycle.SetPolarInterpolationStatus(true);
            csC = Cos(nc.A.v);
            snC = -Sin(nc.A.v);
        }else {          // Switch off
            nc.Block.Out();
            nc.WriteLine("TRANSMIT");
            Cycle.SetPolarInterpolationStatus(true);
            nc.A.RestoreDefaultValue(false);
        }
    }else if (cmd.InterpType == 9022) {// Cylindrical interpolation
        if (cmd.IsOn) { // Switch on
            nc.Block.Out();
            nc.WriteLine("TRACYL(" + Str(2*cld[3]) + ")");
            Cycle.SetCilindInterpolationStatus(true);
        }else{             // Switch off
            nc.Block.Out();
            nc.WriteLine("TRAFOOF");
            //Output "TMCOFF"
            Cycle.SetCilindInterpolationStatus(false);
            nc.A.RestoreDefaultValue(false);
        }
    }
}
```

See also

- CLData access model

LOADTL - Tool loading

Command caption

How the command appears in the CLData command list:

```
LOADTL N n, Tool changed 1, X x, Y y, Z z, D d, M m, K k, L l, P p, A a, R r, H h,
RC rc,
       PLANE XY(33)|YZ(37)|XZ(41)|YX(133)|ZY(137)|ZX(141),
       DUR d2, HID hid, NX nx, NY ny, NZ nz, NW nw, RevolverID rid
```

LOADTL command loads tool of the specified in the first parameter number. Use LOADTL command to form NC-program block that move the tooling turret or select the tool from the tool magazine. Also this command is used to form NC-program frames that move the tool to the safe tool changing position.

Access from sppx

Handler: `program Loadtl`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
n	CLD[1]	CLD.N	Tool number
x y z	CLD[2] CLD[3] CLD[4]	CLD.X CLD.Y CLD.Z	Tool offset along axes X, Y, Z relative to tool fixing point (flange coordinate system).
d	CLD[5]	CLD.D	Tool diameter.
m	CLD[6]	CLD.M	Length corrector number. If the tooling point is the tool end then M < 0.
k	CLD[7]	CLD.K	Radius corrector number. If the tooling point is the tool end then K < 0.

Parameter	CLD array	CLD name	Description
L	CLD[8]	CLD.L	Tool length.
P	CLD[9]	CLD.P	Tool width.
a	CLD[10]	CLD.A	Tool approach angle (for external pocketing 180<a<0, for internal pocketing 0<a<90).
r	CLD[11]	CLD.R	Nose radius.
h	CLD[12]	CLD.H	Tool height.
rc	CLD[13]	CLD.Rc	Radius of cylinder part of axial tool.
XY(33) YZ(37) XZ(41) YX(133) ZY(137) ZX(141)	CLD[14]		Tool plane. A plane normal to the tool axis for axial tool. Machining plane for the lathe tool. 33 – XY, 37 – YZ, 41 – XZ, 133 – YX, 137 – ZY, 141 – ZX.
	CLD[15] CLD[16]		Reserved.
d2	CLD[17]	CLD.Dur	Tool durability (minutes). 0 – ignored.
HID	CLD[18]	CLD.HID	Tool holder block id (number)
nx ny nz nw	CLD[19] CLD[20] CLD[21] CLD[22]	CLD.NX CLD.NY CLD.NZ CLD.NW	The values that define orientation of the tool relative to the anchor point (the flange coordinate system). Depending on the system settings (defined in the machine

Parameter	CLD array	CLD name	Description
			schema) they can be either rotation angles around the coordinate axes X, Y and Z in degrees (Euler angles A = NX, B = NY, C = NZ), either the coefficients of the quaternion q = (NX, NY, NZ, NW).

Access by parameter name (helpers)

Parameters available through the `cmd` operator (the same in sppx and .NET):

TCLDLoadTool: ComplexType	Tool loading command
ToolChanged: bool (integer)	cmd.Int["ToolChanged"] - ToolChanged is a flag that determines whether a tool change is made at this location relative to the previous operation. If it has a value of 1 (true), it means that the tool in this operation is different from the tool in the previous operation. If the flag has a value of 0 (false), it means that the tools in the current and previous operations are the same and there is no tool change. "Skip LoadTL if no tool change" option in the settings window depend on this flag and postprocessor can skip LoadTL procedure calling.
ToolID: Integer	cmd.Int["ToolID"] - Tool identifier (number)
RadCorrNum: Integer	cmd.Int["RadCorrNum"] - Radius corrector number
LenCorrNum: Integer	cmd.Int["LenCorrNum"] - Length corrector number
HolderID: Integer	cmd.Int["HolderID"] - Tool holder identifier (number)
RevolverID: String	cmd.Str["RevolverID"] - String identifier of the turret head

Example (from the Fanuc (30i)_Mill postprocessor):

```
program LoadTl
  OutBlock

  H = abs(cld[6]); H@ = H

  GFunc = 53; GFunc@ = MaxReal
  Z = 0; Z@ = MaxReal
  OutBlock
  GFunc = 53; GFunc@ = MaxReal
  X = 0; X@ = MaxReal
  Y = 0; Y@ = MaxReal
  OutBlock

  M = 06; M@ = M-1
  Tool = cld[1]; Tool@ = MaxReal    ! filling Tool register
  FormBlock
  if NextCode=CodeOfCmd("Comment") then begin
    s$ = CLDFile[CLDFile.CurrentFile].Cmd[CLDFile.CurrentCmd+1].Data
    call TransStr(s$)
    OutStr$ = OutStr$ + " (" + s$ + ")"
  end
  Output OutStr$
  WasLoadTool = 1

  CoordSys = WCSNumber
  CoordSys@ = MaxReal
  OutBlock

  X = MaxReal; X@ = X
  Y = MaxReal; Y@ = Y
  Z = MaxReal; Z@ = Z
!  A = MaxReal; A@ = A
!  B = MaxReal; B@ = B
!  C = MaxReal; C@ = C
end
```

Access from .NET

Handler:

```
public override void OnLoadTool(ICLDLoadToolCommand cmd, CLDArray cld)
```

Inheritance: `ICLDLoadToolCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.ToolChanged`	bool	True if the tool changed since the previous operation; false if the operation uses the same tool.
`cmd.Number`	int	The tool number.
`cmd.LCorNum`	int	First corrector number (length corrector) of the tool.
`cmd.RCorNum`	int	Second corrector number (radius corrector) of the tool.
`cmd.ConnectorIndex`	int	Index of the tool connector inside the machine schema.
`cmd.RevolverID`	string	Turret-head identifier for turret machines; tool-connector identifier for other machine types.
`cmd.Overhang`	`TInpLocation`	Tooltip overhang (X, Y, Z + orientation angles) relative to a spindle; the orientation type depends on the machine schema (Euler angles, normal vector or quaternion).
`cmd.Plane`	`CLDPlaneType`	Tool orientation plane.
`cmd.Durability`	double	Tool durability in minutes.
`cmd.Geom`	`IToolGeometry`	Main geometrical parameters of the tool: `D` (cutting diameter), `R` (tip rounding radius), `Rc`

Property	Type	Description
		(rounding radius next to the cylindrical part), `L` (length), `P` (width), `A` (angle), `H` (height of the shaped part).

Example (from the Sinumerik 840D mill postprocessor):

```
public override void OnLoadTool(ICLDLoadToolCommand cmd, CLDArray cld)
{
    nc.Tool.Show(cmd.Number);
    nc.DTool.Reset(Abs(cmd.LCorNum));
    nc.OutText("M6");
}
```

See also

- CLData access model

MULTIARC - Multi-axis circle movement

Command caption

How the command appears in the CLData command list:

```
MULTIARC EPX ep.X, EPY ep.Y, EPZ ep.Z, EPRX ep.Rx, EPRY ep.Ry, EPRZ ep.Rz,
EPRW ep.Rw,
         MPX mp.X, MPY mp.Y, MPZ mp.Z, MPRX mp.Rx, MPRY mp.Ry, MPRZ mp.Rz,
MPRW mp.Rw
```

MULTIARC movement command defines a circular arc passing through three points with a predetermined orientation of the tool in space for each of these points. The figure below SP - starting position in which the tool was before running the command, MP - an intermediate position of the tool at the arc, EP - end position of the tool.

Access from sppx

Handler: `program MultiArc`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
ep.X ep.Y ep.Z	CLD[1] CLD[2] CLD[3]	CLD.EPX CLD.EPY CLD.EPZ	Cartesian coordinates X, Y and Z of the arc end point
ep.Rx ep.Ry ep.Rz ep.Rw	CLD[4] CLD[5] CLD[6] CLD[7]	CLD.EPNX CLD.EPNY CLD.EPNZ CLD.EPNW	The parameters defining the orientation of the tool at the end point of the arc relative to the current coordinate system (fields RX, RY, RZ and RW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal

Parameter	CLD array	CLD name	Description
			vector n = (RX, RY, RZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = RX, B = RY, C = RZ. Quaternion q = (RX, RY, RZ, RW).
mp.X mp.Y mp.Z	CLD[8] CLD[9] CLD[10]	CLD.MPX CLD.MPY CLD.MPZ	Cartesian coordinates X, Y and Z of the arc intermediate point
mp.Rx mp.Ry mp.Rz mp.Rw	CLD[11] CLD[12] CLD[13] CLD[14]	CLD.MPNX CLD.MPNY CLD.MPNZ CLD.MPNW	The parameters defining the orientation of the tool at the intermediate point of the arc relative to the current coordinate system (fields RX, RY, RZ and RW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (RX, RY, RZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = RX, B =

Parameter	CLD array	CLD name	Description
			RY, C = RZ. Quaternion q = (RX, RY, RZ, RW).

Access by parameter name (helpers)

Parameters available through the `cmd` operator (the same in sppx and .NET):

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
EndPos: ComplexType	cmd.Ptr["EndPos"] - structure whose fields identify the location of tool at the end point of the circle.
Axes: Array, Key="AxisID"	cmd.Ptr["EndPos.Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["EndPos.Axes"].Item[Index] or cmd.Ptr["EndPos.Axes(AxisName)"] - Separate element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["EndPos.Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["EndPos.Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["EndPos.Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
	orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["EndPos.Pos5D.X"] - X cartesian coordinate of the tool tip position.
Y: Double	cmd.Flt["EndPos.Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["EndPos.Pos5D.Z"] - Z cartesian coordinate of the tool tip position.
NX: Double	cmd.Flt["EndPos.Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["EndPos.Pos5D.NY"] - depending on the system configuration may be the NY component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["EndPos.Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["EndPos.Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
	quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["EndPos.MachineStateFlags"].Bit[i], i= (0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of these solutions. For a standard six-axis articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.
MidPos: ComplexType	cmd.Ptr["MidPos"] - structure whose fields identify the location of tool at the intermediate point of the circle.
Axes: Array, Key="AxisID"	cmd.Ptr["MidPos.Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["MidPos.Axes"].Item[Index] or cmd.Ptr["MidPos.Axes(AxisName)"] - Separate

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
	element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["MidPos.Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["MidPos.Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["MidPos.Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["MidPos.Pos5D.X"] - X cartesian coordinate of the tool tip position.
Y: Double	cmd.Flt["MidPos.Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["MidPos.Pos5D.Z"] - Z cartesian coordinate of the tool tip position.

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
NX: Double	cmd.Flt["MidPos.Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["MidPos.Pos5D.NY"] - depending on the system configuration may be the NY component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["MidPos.Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["MidPos.Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["MidPos.MachineStateFlags"].Bit[i], i= (0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of these solutions. For a standard six-axis articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible

TCLDMultiArc: ComplexType	Moving in a circular arc command defined by three points in space.
	solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.
Time: Double	The time of the movement in minutes.

The list of coordinates passed through CLData is defined by the CAM system machine scheme.

Example (from the Newker robot postprocessor):

```
program MultiArc

 !========================== MidPoint
================================================================
  I = CLD.MPX;  J = CLD.MPY;  K = CLD.MPZ
  I@ = MaxReal; J@ = MaxReal; K@ = MaxReal

!========================== EndPoint
================================================================
  X = CLD.EPX;  Y = CLD.EPY;  Z = CLD.EPZ
  A = CLD.EPNX; B = CLD.EPNY; C = CLD.EPNZ;
  X@=MaxReal; Y@=MaxReal; Z@=MaxReal
  A@=MaxReal; B@=MaxReal; C@=MaxReal;
!================================================================================================
==
  call A1_A6(19, CLD.EndAxesCount)
  A1@=maxreal; A2@=maxreal; A3@=maxreal; A4@=maxreal; A5@=maxreal; A6@=maxreal;

  Formblock
  output "{"+outstr$+"}"

  Speed@=Maxreal
  Formblock

  LineN = LineN + 1; LineN@ = LineN
  output "N"+str(LineN)+" MOVC " + outstr$ + " PL=1.0" + ";Arc"
```

```
end
```

Access from .NET

Handlers:

```
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
group handler - before any motion command
public override void OnMultiArc(ICLDMultiArcCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)    //
group handler - after any motion command
```

Inheritance: `ICLDMultiArcCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.EndP`	`ICLDMultiArcPoint`	Finish point of the arc.
`cmd.MidP`	`ICLDMultiArcPoint`	Intermediate point of the arc.
`cmd.Time`	double	Duration of the movement in minutes.
`cmd.Center`	`TInp5DPoint`	Center point and orientation (Euler angles or quaternion) of the arc.
`cmd.R`	double	Arc radius; the sign defines the direction (CW or CCW relative to the arc-plane normal).
`cmd.Ang`	double	Arc angle (always positive).
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z). *(inherited from `ICLDMotionCommand`)*

An arc point (`ICLDMultiArcPoint`) has `P` (cartesian X, Y, Z), `N` (tool orientation: normal vector, Euler angles or quaternion per the machine scheme), `MSFlags` (machine-state flags), `AxesCount`, `Axis` (the list of physical machine-axis values at the point), `Axes` (enumerator), and `HasAxis(string ID)`.

Example (from the Fanuc robot postprocessor - all movement commands share the `Motion` method):

```
public override void OnMultiArc(ICLDMultiArcCommand cmd, CLDArray cld)
{
    Motion(cmd);
}

/// <summary>All movement commands call this method (MultiGOTO, PhysicGoto,
MultiArc, From)</summary>
void Motion(ICLDMotionCommand movement)
{
    state.prevPos = state.Pos;
    if (movement.CmdType==CLDCmdType.MultiArc) {
        ICLDMultiArcCommand cmd = movement as ICLDMultiArcCommand;
        state.Pos = new TInpLocation(cmd.MidP.P, cmd.MidP.N);
        state.J.FillJoints(cmd.MidP.Axes);
        state.Flips.Fill(cmd.Int["MidPos.MachineStateFlags"], state.J);
        int midPointIndex = ls.AddPoint(false, state);
        state.middlePos = state.Pos;

        state.Pos = new TInpLocation(cmd.EndP.P, cmd.EndP.N);
        state.J.FillJoints(cmd.EndP.Axes);
        state.Flips.Fill(cmd.Int["EndPos.MachineStateFlags"], state.J);
        int endPointIndex = ls.AddPoint(false, state);

        CalculateCNT(movement);
        AddMotionLine("C P[" + midPointIndex + "] P[" + endPointIndex + "] " +
Round(state.velocity/60) +
            " mm/sec " + FineOrCnt(state.cntValue) + " ACC" + state.accValue + "
;", false);
    } else {
        ICLDMultiMotionCommand cmd = movement as ICLDMultiMotionCommand;
        state.Pos = new TInpLocation(cmd.EP, cmd.EN);
        state.J.FillJoints(cmd.Axes);
        state.Flips.Fill(cmd.Int["MachineStateFlags"], state.J);
        if (movement.CmdType==CLDCmdType.From) {
            // Output nothing, just remember
            return;
        } else if (movement.CmdType==CLDCmdType.MultiGoto) {
            // Lines
            CalculateCNT(movement);
```

```
            int pointIndex = ls.AddPoint(false, state);
            AddMotionLine("L P[" + pointIndex + "] " + Round(state.velocity/60) + "
mm/sec " + FineOrCnt(state.cntValue) + " ACC" + state.accValue + " ;");
        } else { //PhysicGoto or GoHome
            // Joints
            state.cntValue = smooth.cntValueRapid;
            int pointIndex = ls.AddPoint(true, state);
            if (!state.J.ThereAreE || (state.J.ExtAxesGroup==2)) {
                AddMotionLine("J P[" + pointIndex + "] 10% " +
FineOrCnt(state.cntValue) + " ;");
            } else {
                AddMotionLine("J P[" + pointIndex + "] 10% " +
FineOrCnt(state.cntValue) + " ACC30;");
            }
        }
    }
}
```

See also

- CLData access model

MULTIGOTO - Multi-coordinate movement

Command caption

How the command appears in the CLData command list:

```
MULTIGOTO COUNT 2, MACHINE, X 34.725, Y 32, GEOM X 34.725, Y 32, Z 194, NX 0, NY 0,
NZ 1, NW 0, MSF 0, Time 0.003
```

MULTIGOTO is a universal technology command that is used to pass machine coordinate movement. MULTIGOTO command is used to pass a variable number of coordinates (also called "axes") to postprocessor. These coordinates can be linear movement of tool or table, table or tool rotation or movement of other machine coordinates. Also MULTIGOTO is used to pass miscellaneous technology commands specific for a given machine that can not be passed through more specialized commands. MULTIGOTO command can be used to control common axes like X, Y, Z linear movement, or A, B, C rotatable axes or to control "axes" like chuck jaws, spindle brake, lunet movement and so on. The list of the machine coordinates is defined by the kinematic scheme.

Access from sppx

Handler: `program MultiGOTO`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
X Y Z	CLD[1] CLD[2] CLD[3]	CLD.X CLD.Y CLD.Z	Tooltip spatial coordinates, relative to the active workpiece CS
NX NY NZ NW	CLD[4] CLD[5] CLD[6] CLD[7]	CLD.NX CLD.NY CLD.NZ CLD.NW	Tool frame orientation, relative to the active workpiece CS. The way of orientation definition depend on setting inside Machine schema. It can be Euler angles in different conventions, Tool normal vector,

Parameter	CLD array	CLD name	Description
			quaternion or rotated vector.
MSF	CLD[8]	CLD.MSF	Machine state flags (flips). Actual for the robots and machines that can have multiple configurations of physical axes for same spatial position. See description in the next table.
Time	CLD[9]	CLD.Time	The time of movement.
AxesCount	CLD[10]	CLD.AxesCount	Number of physical machine coordinates affected in this command.
r1	CLD[11]		Index of axis named Axis1Pos in the coordinates list.
Val1	CLD[12]		Value of the machine coordinate Axis1Pos.
r2	CLD[13]		Index of axis named Axis2Pos in the coordinates list.
Val2	CLD[14]		Value of the machine coordinate Axis2Pos.
…	…		…
rN	CLD[2*N+9]		Index of axis named AxisNPos in the coordinates list.

Parameter	CLD array	CLD name	Description
ValN	CLD[2*N+10]		Value of the machine coordinate AxisNPos.

Access by parameter name (helpers)

Parameters available through the `cmd` operator (the same in sppx and .NET):

TCLDMultiGoto: ComplexType	Multi coordinate movement command
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - Separate element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles,

TCLDMultiGoto: ComplexType	Multi coordinate movement command
	ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["Pos5D.X"] - X cartesian coordinate of the tool tip position.
Y: Double	cmd.Flt["Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["Pos5D.Z"] - Z cartesian coordinate of the tool tip position.
NX: Double	cmd.Flt["Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["Pos5D.NY"] - depending on the system configuration may be the NY component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["MachineStateFlags"].Bit[i], i=(0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the

TCLDMultiGoto: ComplexType	Multi coordinate movement command
	possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of these solutions. For a standard six-axis articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.
Time: Double	The time of the movement in minutes.

The list of coordinates passed through CLData of MULTIGOTO command is defined by the CAM system machine scheme.

Use the GMA array to process the MULTIGOTO command. The properties of the GMA array are automatically assigned precalculated values of coordinate movement parameters. The command parameters are also available through the cmd operator.

A simple example of MULTIGOTO command handler with cmd operator using below.

```
program MultiGoto
  if cmd.Ptr["Axes(AxisXPos)"]<>0 then begin
    X = cmd.Flt["Axes(AxisXPos).Value"]
  end
  if cmd.Ptr["Axes(AxisYPos)"]<>0 then begin
    Y = cmd.Flt["Axes(AxisYPos).Value"]
  end
  if cmd.Ptr["Axes(AxisZPos)"]<>0 then begin
    Z = cmd.Flt["Axes(AxisZPos).Value"]
  end
```

```
  if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
    C = cmd.Flt["Axes(AxisCPos).Value"]
  end
  OutBlock
end
```

Access from .NET

Handlers (the specific handler plus the movement-group brackets):

```
public override void OnMultiGoto(ICLDMultiGotoCommand cmd, CLDArray cld)
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)
```

Inheritance: `ICLDMultiGotoCommand : ICLDMultiMotionCommand : ICLDMotionCommand : ICLDCommand` (`ICLDMultiGotoCommand` adds no members of its own; everything below is inherited from `ICLDMultiMotionCommand` / `ICLDMotionCommand`).

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement. *(from `ICLDMotionCommand`)*
`cmd.EN`	`TInpRotation`	End normal or orientation of the movement (normal vector, Euler angles, quaternion, ... per the machine scheme).
`cmd.MSFlags`	`BitArray`	Machine state flags / robot flips that (together with the cartesian coordinates) define the mechanism state in case of redundancy; the meaning of each bit depends on the equipment.

Member	Type	Description
`cmd.Time`	double	Duration of the movement in minutes.
`cmd.AxesCount`	int	The count of axes contained in the `Axis[]` list.
`cmd.Axis`	`ICLDMultiMotionAxesIndexer`	The list of the movement's axes `[0..AxesCount-1]`; indexable by position, by textual `ID`, or by name (`Axis.X/.Y/.Z/.A/.B/.C/.C1/.C2/.A1..A6/.E1..E3`, `null` if absent).
`cmd.Axes`	`IEnumerable`	Axes enumerator, for `foreach`.
`cmd.HasAxis(string ID)`	bool	True if the movement contains an axis with the specified textual ID.

An axis element (`CLDMultiMotionAxis`) has `IntID` (numeric machine-axis id), `ID` (textual machine-axis id as defined in the machine schema), `Value` (final axis value; units - mm/inch/deg/pulse - per the axis definition), and `IsX/IsY/IsZ/IsA/...` tests.

Example (from the Fanuc 30i mill postprocessor):

```
public override void OnMultiGoto(ICLDMultiGotoCommand cmd, CLDArray cld)
{
    foreach (CLDMultiMotionAxis ax in cmd.Axes)
    {
        if (ax.IsX) nc.X.v = ax.Value;
        else if (ax.IsY) nc.Y.v = ax.Value;
        else if (ax.IsZ) nc.Z.v = ax.Value;
        else if (ax.IsA) nc.A.v = ax.Value;
        else if (ax.IsB) nc.B.v = ax.Value;
        else if (ax.IsC) nc.C.v = ax.Value;
    }
    nc.Block.Out();
}
```

See also

- CLData access model

OPSKIP - Optional skipping

> Deprecated. This command is obsolete - the current version of the CAM system does not generate it automatically. It is documented here for postprocessors that still handle legacy CLData.

Command caption

How the command appears in the CLData command list:

```
OPSKIP ON(71) | OFF(72)
```

OPSKIP command activates and deactivates the mode of forming frames of NC-program with optional skip mark. Commonly the symbol `/` is used to mark lines for optional skipping. Marked frames are processed by the CNC-system only if frame skipping option is turned off, otherwise they are ignored.

Access from sppx

Handler: `program OpSkip`

Parameter	CLD array	CLD name	Description
ON(71) OFF(72)	CLD[1]	CLD.onoff	71 (ON) – Activation of optional skipping, 72 (OFF) – Deactivation of optional skipping.

Example (from the Sinumerik (840D) TurnMill postprocessor):

```
program OpSkip
   print "Not marketed "
end
```

See also

- CLData access model

OPSTOP - Auxiliary stop

Command caption

How the command appears in the CLData command list:

```
OPSTOP
```

OPSTOP command is used to form NC-commands of optional breaking of NC-program execution (commonly it's the M01 code). The CNC-system breaks execution on this command only if corresponding execution mode is set, otherwise the NC-program is not halted on this command.

Access from sppx

Handler: `program OpStop`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program OpStop
  if M <> M@ then
    OutBlock
  M = 1      ! M01
  M@ = 0
end
```

Access from .NET

Handler:

```
public override void OnOpStop(ICLDOpStopCommand cmd, CLDArray cld)
```

Inheritance: `ICLDOpStopCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Example - `OnOpStop` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnOpStop(ICLDOpStopCommand cmd, CLDArray cld)
{
```

```
    nc.Block.Out();
    nc.M.Show(1);
    nc.Block.Out();
}
```

See also

- CLData access model

ORIGIN - Original coordinates

Command caption

How the command appears in the CLData command list:

```
ORIGIN X x, Y y, Z z, PPFUN f, N n, A a, B b, C c
```

The ORIGIN command specifies the coordinate system of the subsequent NC code.

Actually there are two modes of the Origin command.

In the first mode the Origin command is used to activate one of the predefined workpiece coordinate systems (G54-G59). The mode is indicated by the `CLD[4]` (`CLD.PPFun`) parameter set to zero. The number of the coordinate system being activated is specified in the `CLD[5]` (`CLD.N`) parameter.

In the second mode the Origin command is used to define a transformation of the workpiece coordinate system (Sinumeric TRANS, ROT, Heidenhain CYCLE7, CYCLE19, PLANE SPATIAL|AXIAL, Fanuc G52, G68 etc). The mode is indicated by the `CLD[4]` (`CLD.PPFun`) parameter set to `PPFun(1079)`. The transformation is defined by shift along the XYZ axes (CLD[1], CLD[2], CLD[3]) and rotation around the same axes (CLD[6], CLD[7], CLD[8]).

Access from sppx

Handler: `program Origin`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
x y z	CLD[1] CLD[2] CLD[3]	CLD.X CLD.Y CLD.Z	Offsets of local system along X, Y and Z axes respectively.
f	CLD[4]	CLD.PPFun	Mode of coordinate system definition: 0 – command specifies one of standard systems; PPFun(1079) – local coordinate system transformation.

Parameter	CLD array	CLD name	Description
n	CLD[5]	CLD.N	The number of chosen standard coordinate system: 54 (G54), 55 (G55), 59 (G59) and so on.
a b c	CLD[6] CLD[7] CLD[8]	CLD.A CLD.B CLD.C	Angles of rotation around X, Y and Z axes respectively. The sequence of rotations around the axes (notation of Euler angles) may vary and is defined in the settings of the machine scheme.

Access by parameter name (helpers)

Parameters available through the `cmd` operator (the same in sppx and .NET):

TCLDOrigin: ComplexType	The command of coordinate system definition
OriginType: Integer	cmd.Int["OriginType"] - Type of coordinate system definition command:0 (SelectStandardLCS) - standard workpiece coordinate system selection command (G54-G59), 1 (TransformLCS) - local coordinate system transformation command.
CSNumber: Integer	cmd.Int["CSNumber"] - The number of the standard workpiece coordinate system: 54 (G54), 55 (G55), 59 (G59) and so on.
MCS: ComplexType	cmd.Ptr["MCS"] - Parameters of the new coordinate system
OriginPoint: ComplexType	cmd.Ptr["MCS.OriginPoint"] - The shift of the original point of the coordinate system
X: Double	cmd.Flt["MCS.OriginPoint.X"] - offset in X

TCLDOrigin: ComplexType	The command of coordinate system definition
Y: Double	cmd.Flt["MCS.OriginPoint.Y"] - offset in Y
Z: Double	cmd.Flt["MCS.OriginPoint.Z"] - offset in Z
RotAngles: ComplexType	cmd.Ptr["MCS.RotAngles"] - The angles of rotation of the coordinate system along the axes (in degrees). The sequence of rotations around the axes (notation of Euler angles) may vary and is defined in the settings of the machine scheme.
A: Double	cmd.Flt["MCS.RotAngles.A"] - angle of rotation around the X axis
B: Double	cmd.Flt["MCS.RotAngles.B"] - angle of rotation around the Y axis
C: Double	cmd.Flt["MCS.RotAngles.C"] - angle of rotation around the Z axis
WCS: ComplexType	cmd.Ptr["WCS"] - Parameters of the new coordinate system. The parameters that defines the transformation on the new coordinate system in relation to the mobile workpiece coordinate system. It includes the geometrical (spatial) coordinates.
OriginPoint: ComplexType	cmd.Ptr["WCS.OriginPoint"] - Shift of coordinate system origin
X: Double	cmd.Flt["WCS.OriginPoint.X"] - offset in X
Y: Double	cmd.Flt["WCS.OriginPoint.Y"] - offset in Y
Z: Double	cmd.Flt["WCS.OriginPoint.Z"] - offset in Z
RotAngles: ComplexType	cmd.Ptr["WCS.RotAngles"] - The angles of rotation of the coordinate system along the axes (in degrees). The sequence of rotations around the axes (notation of Euler angles) may vary and is defined in the settings of the machine scheme.

TCLDOrigin: ComplexType	The command of coordinate system definition
A: Double	cmd.Flt["WCS.RotAngles.A"] - angle of rotation around the X axis
B: Double	cmd.Flt["WCS.RotAngles.B"] - angle of rotation around the Y axis
C: Double	cmd.Flt["WCS.RotAngles.C"] - angle of rotation around the Z axis
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - Array of the "Axes" records. Defines the positions of the real (machine) axes that corresponds to the defined coordinate system. Origin command can contain positions for the few axes.
Axis: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - The item of the Axes array. It contains the position of a machine axis. Access for the array item is available by the index of by the key field. Here AxisName - value of the key field that must be equal to value of field AxisID.
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - Identifier of machine axis for that the position is defined. It is described in the machine schema.
Value: Double	cmd.Flt["Axes(AxisName).Value"] - Axis position.
IsSpatial: Integer	cmd.Int["IsSpatial"] - parameter, that defines the priority what is main or geometrical coordinates (A,B,C) or machine axes. Two values is possible: 0 - machine axes defined by Axes array has the priority.1 - geometrical coordinates defined by the MCS and WCS have the priority.
PositioningMode: Integer	cmd.Int["PositioningMode"] - parameter that defines or this command generate the real motions of the machine axes or defines the local coordinate system inside CNC without any motions.Three value are possible:0 - STAY -

TCLDOrigin: ComplexType	The command of coordinate system definition
	transformation does not generate the real machine motions., 1 - TURN - transformation moves the machine rotary axes only., 2 - MOVE transformation moves all necessary axes to save the same tool tip position in the new coordinate system.

Some controls require definition of local coordinate systems by spatial angles while the other controls support definition of local coordinate systems only by real machine angles. The ORIGIN command includes all the needed information for the both definition methods. The cmd.Ptr["MCS"] and the cmd.Ptr["WCS"] properties contain spatial angles (A, B, C) of the LCS, while the cmd.Ptr["Axes"] array contain the corresponding coordinates of machine rotary axes. The main advantage of the cmd.Ptr["Axes"] property is that it defines the LCS unambiguously while for the cmd.Ptr["WCS"] there are commonly two possible solutions. For the controls requiring spatial angles for the LCS definition the machine angles can be used to resolve ambiguous cases. (See an example below).

Note: spatial angles are specified relative to the local coordinate system being active at the moment of definition of the ORIGIN command.

In case a control supports both methods of LCS definition, you can specify the desired output format of the ORIGIN command in CAM system in the Control parameters settings. In the postprocessor the desired output format is available in the cmd.Int["IsSpatial"] property of the ORIGIN command.

Some controls support various positioning modes of the ORIGIN command. CAM system offers the following LCS positioning modes.

cmd.Int["PositioningMode"]=STAY. The origin command does not move the machine axes.

cmd.Int["PositioningMode"]=TURN. The origin command rotates the machine axes in such a way the tool axis direction becomes aligned with the Z axis of the local coordinate system

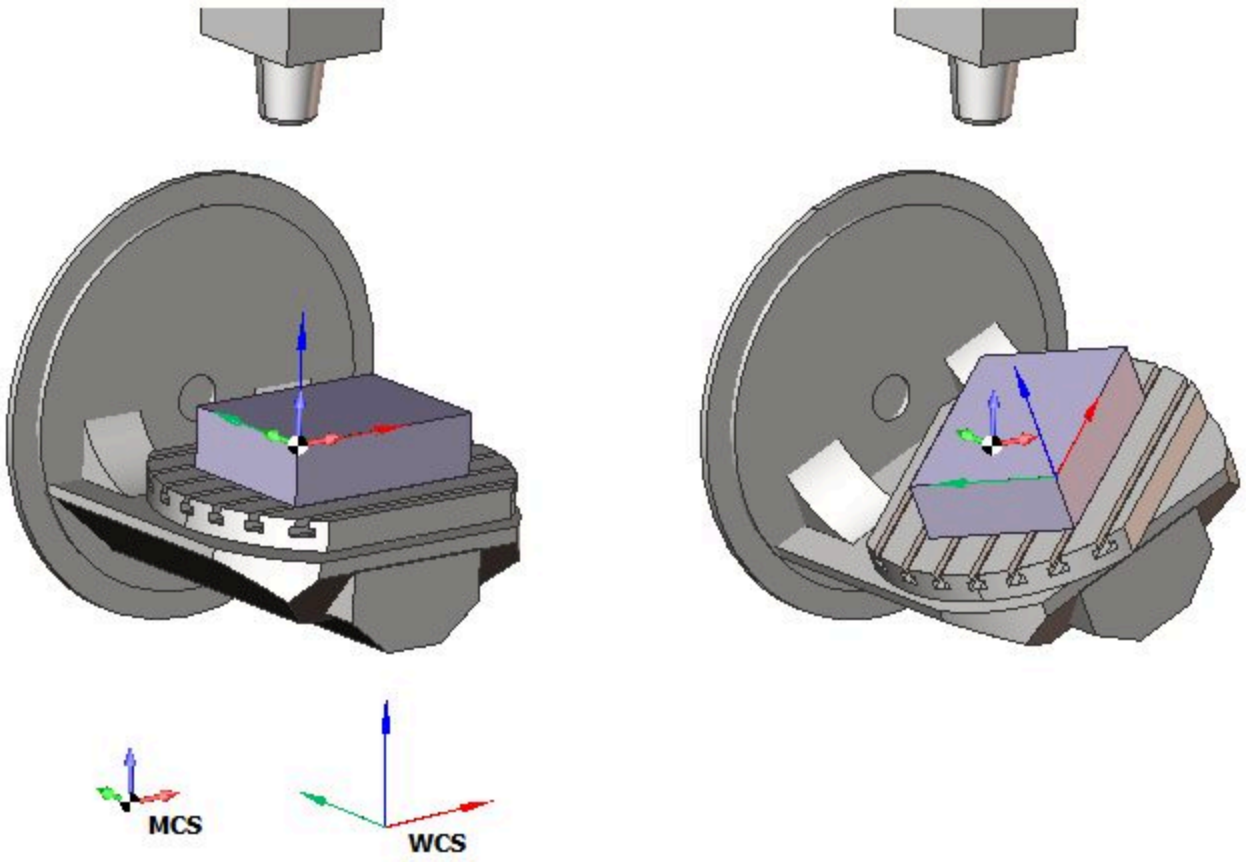
cmd.Int["PositioningMode"]=MOVE. The origin command rotates the machine axes to align the tool axis with the Z axis of the local coordinate system and moves the linear axes in such a way the tool tip position stays the same relative to the workpiece

The controls define a local coordinate system relative to some reference workpiece coordinate system (G54). Depending on the control used you should use either the MCS matrix or the WCS matrix to define a LCS.

Modern controls know about the machine kinematics. On those controls the workpiece coordinate system is rotated together with the rotary table. So on those controls you can define a local coordinate system relative to the rotating workpiece coordinate system. In this case you are happy to use the WCS matrix.

Old controls know nothing about the machine kinematics and can not rotate the workpiece coordinate system together with the rotary table. For those controls you should use the MCS matrix to define a LCS transformation.

At the following figures you can see the difference between the reference coordinate systems for the MCS, and the WCS matrices. Using the WCS matrix is preferable as the WCS matrix is not dependent on the workpiece setup, while the MCS is heavily dependent on it.



CAM system Local CS settings.

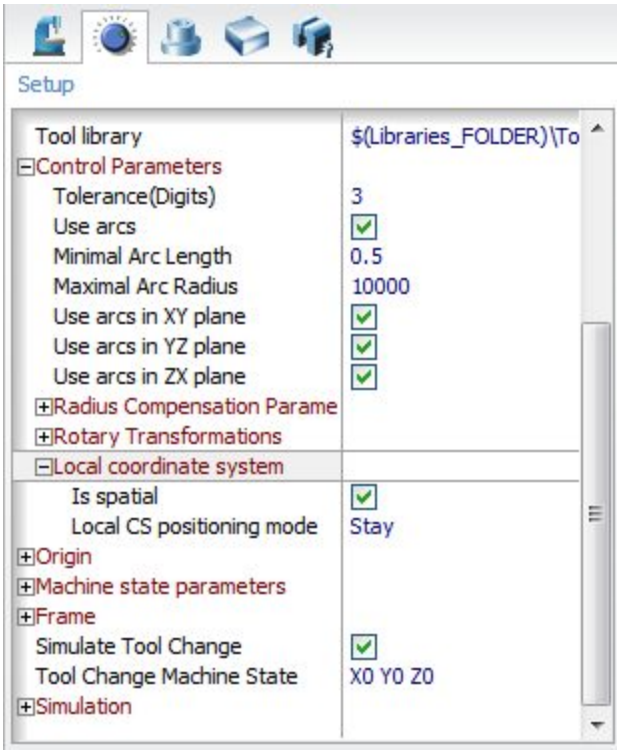
In CAM system the settings of the ORIGIN command are specified in the Machine configuration file.

```
SCType ID="MachineName" type="AbstractMachine"


  ...
  <ControlData>
  ...
  <LocalCS>
  <IsSpatial DefaultValue="True"/>
  <PositioningMode DefaultValue="Stay"/>
  </LocalCS>
  ...
  </ControlData>
```

```
...
<SCType>
```

The options are also available from the GUI in the Control Parameters section of the Machine parameters.



Code samples of the ORIGIN command.

Example - a G54-G59 / G52 datum shift:

```
program Origin
  if cmd.Int["OriginType"]=0 then begin !G54-G59
    Output "G" + cmd.Str["CSNumber"]
  end else begin !G52 X Y Z
    X = cmd.Flt["MCS.OriginPoint.X"]
    Y = cmd.Flt["MCS.OriginPoint.Y"]
    Z = cmd.Flt["MCS.OriginPoint.Z"]
    Output "G52 X" + Str(X) + " Y" + Str(Y) + " Z" + Str(Z)
  end
end
```

Example - a Heidenhain-style CYCL DEF datum shift and working plane:

```
program Origin
  if CLD[4]=0 then begin ! Select standart LCS (G54-G59)
    ! Do nothing for Heidenhain
```

```
  end else if CLD[4]=1079 then begin ! Transform LCS
    Output "CYCL DEF 7.0 DATUM SHIFT"
    Output "CYCL DEF 7.1 X" + Str(cmd.Flt["WCS.OriginPoint.X"])
    Output "CYCL DEF 7.2 Y" + Str(cmd.Flt["WCS.OriginPoint.Y"])
    Output "CYCL DEF 7.3 Z" + Str(cmd.Flt["WCS.OriginPoint.Z"])
    if (abs(cmd.Flt["WCS.RotAngles.A"])>0.0001) or
    (abs(cmd.Flt["WCS.RotAngles.B"])>0.0001) or
    (abs(cmd.Flt["WCS.RotAngles.C"])>0.0001)
    then begin
      Output "CYCL DEF 19.0 WORKING PLANE"
      if cmd.Int["IsSpatial"]>0 then begin ! Spatial coordinates
        Output "CYCL DEF 19.1 A" + str(cmd.Flt["WCS.RotAngles.A"]) +
        " B" + str(cmd.Flt["WCS.RotAngles.B"]) +
        " C" + str(cmd.Flt["WCS.RotAngles.C"])
      end else begin ! Machine axes
        OutStr$ = "CYCL DEF 19.1"
        if cmd.Ptr["Axes(AxisBPos)"]<>0 then begin
          OutStr$ = OutStr$ + " B" + str(cmd.Flt["Axes(AxisBPos).Value"])
        end
        if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
          OutStr$ = OutStr$ + " C" + str(cmd.Flt["Axes(AxisCPos).Value"])
        end
        Output OutStr$
      end
    end
  end
end
```

Access from .NET

Handlers - besides the base `OnOrigin`, the framework calls `OnWorkpieceCS` for the G54-G59 case and `OnLocalCS` for a local-CS transform; ORIGIN is also a movement command, so the movement-group brackets fire:

```
public override void OnOrigin(ICLDOriginCommand cmd, CLDArray cld)         //
base dispatcher
public override void OnWorkpieceCS(ICLDOriginCommand cmd, CLDArray cld)    //
G54-G59
public override void OnLocalCS(ICLDOriginCommand cmd, CLDArray cld)        // local
CS transform
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)
```

Inheritance: `ICLDOriginCommand : ICLDMultiMotionCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.OriginType`	`CLDOriginType`	Standard workpiece CS selection (G54-G59) or a specific local CS switch (G68.2, PLANE SPATIAL, Cycle800).
`cmd.IsWorkpieceCS` `cmd.IsLocalCS`	bool	True if this is a workpiece-CS selection / a local-CS switch command.
`cmd.IsOn` `cmd.IsOff`	bool	True if this is a CS switch-ON / switch-OFF command.
`cmd.CSNumber`	double	Number of the standard workpiece CS to activate.
`cmd.MCS`	`TInpLocation`	Cartesian coordinates of the new CS relative to the machine base.
`cmd.WCS`	`TInpLocation`	Cartesian coordinates of the new CS relative to the workpiece (may rotate with it).
`cmd.QMCS` `cmd.QWCS`	`TInpQuaternion`	Quaternion orientation of the new CS relative to the machine base / the workpiece.
`cmd.Convention`	`TInpComplexRotationConvention`	The Euler-angle convention used inside MCS and WCS.
`cmd.IsSpatial`	bool	True if cartesian coordinates should be preferred over physical machine axes for defining the new CS.

Property	Type	Description
`cmd.PositioningMode`	`CLDOriginPositionMode`	How machine axes change when switching CS: `Stay`, `Turn` or `Move`.

Plus the inherited `ICLDMultiMotionCommand` members (`EN`, `MSFlags`, `Time`, `AxesCount`, `Axis`, `Axes`, `HasAxis`) and `EP` - see MULTIGOTO for their descriptions.

Example - `OnWorkpieceCS` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnWorkpieceCS(ICLDOriginCommand cmd, CLDArray cld)
{
    nc.GWCS.v = cmd.CSNumber;    // G54..G59
}
```

Example - `OnLocalCS` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnLocalCS(ICLDOriginCommand cmd, CLDArray cld)
{
    if (cmd.IsOn) {
        nc.GLCS.Show(68.2);
        nc.X.Show(cmd.WCS.P.X);
        nc.Y.Show(cmd.WCS.P.Y);
        nc.Z.Show(cmd.WCS.P.Z);
        nc.I.Show(cmd.WCS.N.A);
        nc.J.Show(cmd.WCS.N.B);
        nc.K.Show(cmd.WCS.N.C);
        nc.Block.Hide(nc.GInterp);
        nc.Block.Out();
        nc.Block.Reset(nc.X, nc.Y, nc.Z);
        nc.OutWithN("G53.1");
    } else {
        nc.GLCS.Show(69);
        nc.Block.Out();
    }
}
```

See also

- CLData access model

PALETA - Palette changing

> Deprecated. This command is obsolete - the current version of the CAM system does not generate it automatically. It is documented here for postprocessors that still handle legacy CLData.

Command caption

How the command appears in the CLData command list:

```
PALETA N n
```

Use PALETA command to form NC-program frames that control palette change. The new palette number is passed as the command parameter.

Access from sppx

Handler: `program Paleta`

Parameter	CLD array	CLD name	Description
n	CLD[1]	CLD.N	Palette number

See also

- CLData access model

PARTNO - Part number

Command caption

How the command appears in the CLData command list:

```
PARTNO "<CAM system project name>"
```

PARTNO is the first command in the list of technology commands of a project. Use PARTNO command to form NC-program initialization block. PARTNO command puts the project name into the CLDATA$ variable.

Access from sppx

Handler: `program PartNo`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PartNo
! Initialization of global variables
  MaxReal = 99999.999
  XT_ = 0;  FROMX_ = 0
  YT_ = 0;  FROMY_ = 0
  ZT_ = 0;  FROMZ_ = 0

  KorEcv = 40; KorEcv@=KorEcv
  KorDl  = 49; KorDl@=KorDl
  Cycle  = 80; Cycle@=Cycle
  Plane  = -17; Plane@=Plane
  GSpindl = 97; GSpindl@ = GSpindl
  CoordSys = 54; CoordSys@ = CoordSys; WCSNumber = 54
  PG = 0; PG@ = PG
  YAddress = "Y"
  IsRotaryInterpolation = 0
  IsMultiaxisInterpolation = 0
  CylR = 0
  Pi = 3.14159265358979323846264338327795
  Deg2Rad = Pi / 180
  Rad2Deg = 180 / Pi
  CirclesThroughRadius = 0
  call check_Euler_angles_types
  IsLocalCS = 0
  G68_2_X = MaxReal
  G68_2_Y = MaxReal
  G68_2_Z = MaxReal
```

```
  G68_2_I = MaxReal
  G68_2_J = MaxReal
  G68_2_K = MaxReal
  HasAxisA = 0
  HasAxisB = 0
  HasAxisC = 0

! Constsnt cycles variables
  CycleOn = 0          ! 0 - cycle Off, 1 - cycle On
  KodCycle = 0         ! constant cycle code
  Za = 0               ! work depth
  Zf = 0               ! safe level
  ZP_ = 0               ! reverse move level
  Zl = 0               ! depth of one drilling step
  Zi = 0               ! transitional value
  Dwell = 0            ! pause in constsnt cycles
  Fr = 0               ! work FEED_
! Set machene functions by default
  X = MaxReal; X@=X
  Y = MaxReal; Y@=Y
  Z = MaxReal; Z@=Z
  A = MaxReal; A@=A
  B = MaxReal; B@=B
  C = MaxReal; C@=C
  S = 0; S@=S
  SpDir = 5
  SpRPM = 0
  DrillPause = MaxReal; DrillPause@ = DrillPause;
  QStep = MaxReal; QStep@ = QStep;
  Zcycle=Maxreal; Zcycle@=Zcycle
  P=0;Q=0
  D_=0; H_=0;             ! values D, H
  RapFR  = 2000
  INTERP_ = 100;  INTERP_@ = 100
  Msp = 5; Msp@ = 5     ! spindle off
  Mc=9; Mc@=Mc;
  Zcycle=Maxreal
  OperationIsLathe = 2
  BlkN=100

! Creating NC heading
  output "%"
  MainProgramID = 5
  call GetProgNumberFromName(NCName$,MainProgramID)
  !Input "Input program number:", MainProgramID
!  ProgNum = MainProgramID; ProgNum@ = MaxReal
```

```
  FormBlock
  s$ = NCName$
  call CheckProgramName(s$)
  output OutStr$ + UPCASE(s$)

  output ""
  Output "( GENERATED BY CAM )"
  Output "( DATE: " + CurDate + " )"
  Output "( TIME: " + CurTime + " )"

  output ""
  call PrintAllTools
  output ""
  Output "G00G21G40G49G69G80G90G17"
  output "<StartGoHome>"

  OutBlock
end
```

Access from .NET

Handlers:

```
public override void OnPartNo(ICLDPartNoCommand cmd, CLDArray cld)
public override void OnStartProject(ICLDProject prj)   // project-level, raised
on PARTNO
```

Inheritance: `ICLDPartNoCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.ProjectName`	`string`	Returns the project's file name without path and extension.

Example - `OnStartProject` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnStartProject(ICLDProject prj)
{
```

```
    probingCycle.Init();

    mainNC = new NCFile();
    nc = mainNC;
    nc.OutputFileName = Settings.Params.Str["OutFiles.NCFileName"];
    // Log.Info("Output file name: " + nc.OutputFileName);
    nc.ProgNumber = Settings.Params.Int["OutFiles.NCProgNumber"];

    CirclesThroughRadius = Settings.Params.Bol["BlockFormat.CirclesThroughRadius"];

    if (Settings.Params.Bol["BlockFormat.Numbering"])
        nc.BlockN.Show();
    else
        nc.BlockN.Disable();
    if (Settings.Params.Bol["BlockFormat.Spaces"])
        nc.Block.WordsSeparator = " ";
    else
        nc.Block.WordsSeparator = "";

    nc.WriteLine("%");
    nc.WriteLine("O" + Str(nc.ProgNumber));

    nc.WriteLine();
    nc.WriteLine(String.Format("( {0} )", "Generated by CAM"));
    nc.WriteLine(String.Format("( {0} )", "Date: " +
DateTime.Now.ToShortDateString()));
    nc.WriteLine(String.Format("( {0} )", "Time: " +
DateTime.Now.ToShortTimeString()));
    nc.WriteLine();

    PrintAllTools();
    nc.WriteLine();

    nc.Block.Show(nc.GAbsInc, nc.GFeed ,nc.GMeasure, nc.GWCS, nc.GLCS, nc.GPlane,
        nc.GLCompens, nc.GRCompens, nc.GInterp, nc.GCycle);
    nc.Block.Out();
    nc.OutWithN("G53", nc.Z.ToString(0));
    OutG53ABC();
}
```

See also

- CLData access model

PHYSICGOTO - Physical machine axes movement

Command caption

How the command appears in the CLData command list:

```
PhysicGOTO COUNT 2, MACHINE, X 0, Y 0, GEOM X 125.281, Y -32, Z 194, NX 0, NY 0, NZ
1, NW 0, MSF 0, Time 0.013
```

PhysicGOTO is a universal technology command that is used to pass machine coordinate movement. It is similar to the MULTIGOTO command, the difference is as follows. The MULTIGOTO command contains the coordinates of the axes relative to the current workpiece coordinate system (spatial coordinates relative to the G54-G56 or the local coordinate system), and the basic axes interpolation is performed to control the straightness of the path for each of the intermediate points. In ISO code, this corresponds to a linear interpolation commands G01. In robot control system, this corresponds to the type of movement Linear (L, LIN, MOVEL), Control path (CP) motion. In contrast, the PhysicGOTO command contains the physical coordinate values relative to machine zeros for each axis, and the base axes interpolation does not perform control of the intermediate points of the toolpath, and checked only the reachability of the endpoint. In the ISO codes, it corresponds to the G53 move command (SUPA in Sinumerik, M91 in HEIDENHAIN). In the robot control system it corresponds to the movement type Joint move (J, MOVEJ), Point-to-point (PTP) motion.

Access from sppx

Handler: `program PhysicGOTO`

Parameter	CLD array	Description
X Y Z	CLD[1] CLD[2] CLD[3]	Tooltip spatial coordinates, relative to the active workpiece CS
NX NY NZ NW	CLD[4] CLD[5] CLD[6] CLD[7]	Tool frame orientation, relative to the active workpiece CS. The way of orientation definition depend on setting inside Machine schema. It can be Euler angles in different conventions, Tool normal vector, quaternion or rotated vector.

Parameter	CLD array	Description
MSF	CLD[8]	Machine state flags (flips). Actual for the robots and machines that can have multiple configurations of physical axes for same spatial position. See description in the next table.
Time	CLD[9]	The time of movement.
AxesCount	CLD[10]	Number of physical machine coordinates affected in this command.
r1	CLD[11]	Index of axis named Axis1Pos in the coordinates list.
Val1	CLD[12]	Value of the machine coordinate Axis1Pos.
r2	CLD[13]	Index of axis named Axis2Pos in the coordinates list.
Val2	CLD[14]	Value of the machine coordinate Axis2Pos.
…	…	…
rN	CLD[2*N+9]	Index of axis named AxisNPos in the coordinates list.
ValN	CLD[2*N+10]	Value of the machine coordinate AxisNPos.

Parameters available through the cmd operator

TCLDPhysicGoto: ComplexType	Multiaxis physical machine coordinate movement command
Axes: Array, Key="AxisID"	cmd.Ptr["Axes"] - An array of structures such as Axis. One command may include movement along several axes, the position of each is stored in this array.
Axis: ComplexType	cmd.Ptr["Axes"].Item[Index] or cmd.Ptr["Axes(AxisName)"] - Separate element of the Axes array. Contains information about moving on one machine axis. Access to the array elements can either by index or by key field. Here AxisName - the key field value, which must match the AxisID field value.
AxisID: String	cmd.Str["Axes(AxisName).AxisID"] - The identifier of the machine axis, for which the specified new position. Determined by the machine schema.
Value: Double	cmd.Flt["Axes(AxisName).Value"] - The new position of the machine axis, in which it moves.
Pos5D: ComplexType	cmd.Ptr["Pos5D"] - structure containing the coordinates that define the final position of the tool in space. Determines not only the position of the tuning point of the tool (the fields X, Y and Z - cartesian coordinates), but also the orientation of the tool relative to the current coordinate system (fields NX, NY, NZ and NW). Depending on the configuration of the system (defined in the scheme of machine), the tool orientation can be specified by: Normal vector n = (NX, NY, NZ), Spatial angles of rotation around the coordinate axes - the Euler angles (notation of Euler angles, ie the sequence of rotations is also specified in the settings) A = NX, B = NY, C = NZ. Quaternion q = (NX, NY, NZ, NW).
X: Double	cmd.Flt["Pos5D.X"] - X cartesian coordinate of the tool tip position.

TCLDPhysicGoto: ComplexType	Multiaxis physical machine coordinate movement command
Y: Double	cmd.Flt["Pos5D.Y"] - Y cartesian coordinate of the tool tip position.
Z: Double	cmd.Flt["Pos5D.Z"] - Z cartesian coordinate of the tool tip position.
NX: Double	cmd.Flt["Pos5D.NX"] - depending on the system configuration may be the NX component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NY: Double	cmd.Flt["Pos5D.NY"] - depending on the system configuration may be the NY component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NZ: Double	cmd.Flt["Pos5D.NZ"] - depending on the system configuration may be the NZ component of the tool normal vector, the rotation angle around one of the coordinate axis, or the coefficient of the quaternion.
NW: Double	cmd.Flt["Pos5D.NW"] - if you have chosen a method of orientation tool in the form of a quaternion is a fourth coefficient of quaternion.
MachineStateFlags: Integer	cmd.Ptr["MachineStateFlags"].Bit[i], i=(0..31). Integer number which is a bit field of 32 bits from 0 to 31. If there are several variants for the location of the machine axes, providing a specified position of the tool in space, every bit of this field determines the selection of one of the possible solutions. For example, a five-axis machine with A and C axes for most of the positions of the tool has two possible solutions - with a positive A-axis and with negative value of the A axis. Zero bit of this field will choose one of

TCLDPhysicGoto: ComplexType	Multiaxis physical machine coordinate movement command
	these solutions. For a standard six-axis articulated robot many of the tool positions in space can be provided with different combinations of three of its key joints positions (the base, elbow and wrist joints) giving a total of 8 possible solutions. In this case, the zero bit in this field will determine the position of the base, the first bit - the position of the elbow, and the third bit will be set the position of the robot's wrist. Thus, the meaning of each of this bits is determined entirely by the used machine schema. When using machines that do not need to use these flags, it is possible to disable the output of this parameter to CLData in the machine scheme.
Time: Double	The time of the movement in minutes.

The list of coordinates passed through CLData of PHYSICGOTO command is defined by the CAM system machine scheme.

Use the GMA array to process the PHYSICGOTO command. The properties of the GMA array are automatically assigned precalculated values of coordinate movement parameters. The command parameters are also available through the cmd operator.

A simple example of PHYSICGOTO command handler with cmd operator using below.

```
program PhysicGoto
  if cmd.Ptr["Axes(AxisXPos)"]<>0 then begin
    X = cmd.Flt["Axes(AxisXPos).Value"]
  end
  if cmd.Ptr["Axes(AxisYPos)"]<>0 then begin
    Y = cmd.Flt["Axes(AxisYPos).Value"]
  end
  if cmd.Ptr["Axes(AxisZPos)"]<>0 then begin
    Z = cmd.Flt["Axes(AxisZPos).Value"]
  end
  if cmd.Ptr["Axes(AxisAPos)"]<>0 then begin
    A = cmd.Flt["Axes(AxisAPos).Value"]
  end
  if cmd.Ptr["Axes(AxisCPos)"]<>0 then begin
    C = cmd.Flt["Axes(AxisCPos).Value"]
```

```
  end
  if (X<>X@) or (Y<>Y@) or (Z<>Z@) or (A<>A@) or (C<>C@) then begin
    Interp_ = 53; Interp_@ = 0; ! G53
    OutBlock
  end
end
```

Access from .NET

Handlers:

```
public override void OnBeforeMovement(ICLDMotionCommand cmd, CLDArray cld)   //
group handler - before any motion command
public override void OnPhysicGoto(ICLDPhysicGotoCommand cmd, CLDArray cld)
public override void OnAfterMovement(ICLDMotionCommand cmd, CLDArray cld)   // group
handler - after any motion command
```

Inheritance: `ICLDPhysicGotoCommand : ICLDMultiMotionCommand : ICLDMotionCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.EN`	`TInpRotation`	End normal or orientation of the movement in cartesian coordinate system. It can be normal vector, euler angles, quaternion etc. depend on the current machine schema settings.
`cmd.MSFlags`	`BitArray`	Machine state flags or robot flips which can help to define (together with cartesian coordinates) the specific state of the mechanism in case of redundancy. Which bit is responsible for which flag

Member	Type	Description
		depends on the type and configuration of the equipment.
`cmd.Time`	`double`	Duration of the movement in minutes.
`cmd.AxesCount`	`int`	The count of axes that the movement contains inside the Axis[] list.
`cmd.Axis`	`ICLDMultiMotionAxesIndexer`	The list of the movement's axes [0..(AxesCount-1)].
`cmd.Axes`	`IEnumerable`	Axes' enumerator object for the movement to be possible to use covenient foreach syntax.
`cmd.HasAxis(...)`	`bool`	Returns "true" if the movement contains axis with the speciefed textual ID. Otherwise returns "false".
`cmd.EP`	`TInp3DPoint`	End point (X, Y, Z) of the movement

Example - `OnPhysicGoto` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnPhysicGoto(ICLDPhysicGotoCommand cmd, CLDArray cld)
{
    foreach(CLDMultiMotionAxis ax in cmd.Axes) {
        if (ax.IsX) {
            nc.X.Show(ax.Value);
        } else if (ax.IsY) {
            nc.Y.Show(ax.Value);
        } else if (ax.IsZ) {
            nc.Z.Show(ax.Value);
        } else if (ax.IsA) {
            nc.A.Show(ax.Value);
        } else if (ax.IsB)  {
            nc.B.Show(ax.Value);
        } else if (ax.IsC) {
            nc.C.Show(ax.Value);
        }
```

```
    }
    if (nc.X.Changed || nc.Y.Changed || nc.Z.Changed || nc.A.Changed || nc.B.Changed
|| nc.C.Changed) {
        nc.GHome.Show(53);
        nc.Block.Out();
    }
}
```

See also

- CLData access model

PLANE - Working plane

Command caption

How the command appears in the CLData command list:

```
PLANE XY(33) | YZ(37) | XZ(41) | YX(133) | ZY(137) | ZX(141)
```

PLANE command is used to define active machining plane. Commonly this command is processed into G17, G18, G19 G-codes. Active plane is used in circular interpolation (it defines plane of arcs), to implement tool radius offset, to define hole machining cycles etc.

Access from sppx

Handler: `program Plane`

Parameter	CLD array	CLD name	Description
XY(33) YZ(37) XZ(41) YX(133) ZY(137) ZX(141)	CLD[1]	CLD.Plane	Plane code: 33 – XY, 37 – YZ, 41 – XZ, 133 – YX, 137 – ZY, 141 – ZX.

To create ISO postprocessor use ISO.G mask, which is assigned the plane code automatically:

CLD item value	ISO value
CLD[1] = 33	ISO.G = 17
CLD[1] = 37	ISO.G = 19
CLD[1] = 41	ISO.G = 18

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Plane
  case cld[1] of
    33: Plane = 17;      ! XY
    37: Plane = 19;      ! YZ
    41: Plane = 18;      ! XZ
```

```
    133: Plane = -17;      ! XY
    137: Plane = -19;      ! YZ
    141: Plane = -18;      ! XZ
  end
  if (Plane<>Plane@) and (abs(Plane)=abs(Plane@)) then
    Plane@ = Plane
!  if Plane<>Plane@ then
!    OutBlock
  case abs(Plane) of
    17: begin
      regZPlane = Rgs["Z"].Num
      regZCycle = Rgs["ZCYCLE"].Num
    end
    18: begin
      regZPlane = Rgs["Y"].Num
      regZCycle = Rgs["YCYCLE"].Num
    end
    19: begin
      regZPlane = Rgs["X"].Num
      regZCycle = Rgs["XCYCLE"].Num
    end
    else begin
      regZPlane = Rgs["Z"].Num
      regZCycle = Rgs["ZCYCLE"].Num
    end
  end
end
```

Access from .NET

Handler:

```
public override void OnPlane(ICLDPlaneCommand cmd, CLDArray cld)
```

Inheritance: `ICLDPlaneCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.Plane`	`CLDPlaneType`	Contains exact plane type. 6 standard values are possible: XY, YZ, ZX, InvXY, InvYZ, InvZX. Possible values: `XY` (17) - Standard plane XY or G17 `YZ` (19) - Standard plane YZ or G19 `ZX` (18) - Standard plane ZX or G18 `InvXY` (-17) - Standard plane opposite to XY plane (-17) `InvYZ` (-19) - Standard plane opposite to YZ plane (-19) `InvZX` (-18) - Standard plane opposite to ZX plane (-18)
`cmd.PlaneGCode`	`int`	Contains standard ISO G-code number for the plane type. 6 values are possible: 17, 18, 19, -17, -18, -19.
`cmd.PlaneSign`	`int`	Returns "+1" for the planes XY, YZ, ZX. Returns "-1" for the planes InvXY, InvYZ and InvZX.

Example - `OnPlane` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnPlane(ICLDPlaneCommand cmd, CLDArray cld)
{
    nc.GPlane.v = cmd.PlaneGCode;
    planeSign = cmd.PlaneSign;
    switch (cmd.Plane) {
        case CLDPlaneType.XY:
        case CLDPlaneType.InvXY:
            planeZIndex = 3;
            nc.PlaneZReg = nc.Z;
            break;
        case CLDPlaneType.ZX:
        case CLDPlaneType.InvZX:
```

```
            planeZIndex = 2;
            nc.PlaneZReg = nc.Y;
            break;
        case CLDPlaneType.YZ:
        case CLDPlaneType.InvYZ:
            planeZIndex = 1;
            nc.PlaneZReg = nc.X;
            break;
    }
}
```

See also

- CLData access model

POWER - Power adjustment

Command caption

How the command appears in the CLData command list:

```
POWER N n, Value
```

POWER CLData command specifies the percent of a nominal power (defined in the operation parameters of CAM) for the exact active effector (laser, jet nozzle, arc welder etc.) that should be used for the following movements. For example, the CAM system may calculate the value passed by this command as a percentage of the nominal workpiece thickness or a percentage of the thickness of the layer being machined (the closest point in the previous pass).

Access from sppx

Handler: `program Power`

Parameter	CLD array	CLD name	Description
N	CLD[1]	CLD.N	Unique identifier of the device in the CAM whose power we should change
Value	CLD[2]	CLD.Value	The value of a power in percents of some constant value specified either in the parameters of the operation, or implicitly assumed.

Example:

```
program Power
  ! cld[1] - device id, cld[2] - power as a percentage of the nominal value
  NominalCurrent = 120                                  ! nominal current (A) of the
effector, usually taken from operation params
  Output "A" + Str(NominalCurrent * cld[2] / 100)       ! actual current for the
```

```
following movements
end
```

Access from .NET

Handler:

```
public override void OnPower(ICLDPowerCommand cmd, CLDArray cld)
```

Inheritance: `ICLDPowerCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.ID`	`int`	Numeric identifier of the device for which the power is set.
`cmd.Value`	`double`	The value of a power in percents of some constant value specified either in the parameters of the operation, or implicitly assumed.

Example:

```
public override void OnPower(ICLDPowerCommand cmd, CLDArray cld)
{
    // cmd.Value is a percentage of the nominal power of the active effector
    double nominalCurrent = 120;                          // nominal current (A),
usually taken from the operation parameters
    nc.Amperage.Show(nominalCurrent * cmd.Value / 100);   // actual current for the
following movements
    nc.Block.Out();
}
```

See also

- CLData access model

PPFUN - Postprocessor function

Command caption

How the command appears in the CLData command list:

```
PPFUN
PPFUN(500)|STARTSUB(50)|ENDSUB(51)|CALLSUB(52)|REPSTART(53)|REPEND(54)|JUMP(55)|
      TECHINFO(58)|ENDTECHINFO(59)|WEDMConditions(56)|WEDMPassItem(57) {, a} {,
b} ...
```

Postprocessor function PPFUN is a multiple purpose postprocessors generator command. PPFUN command can be used to present different kinds of data and machining parameters. The actual type of PPFUN command is defined by the first parameter of the CLData array – `CLD[1]` (`CLD.SubCode`). Other CLD array parameters (`CLD[2]` through `CLD[257]`) can have various meaning dependinig on the type of the PPFUN command.

Access from sppx

Handler: `program PPFun`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
PPFUN(500) STARTSUB(50) ENDSUB(51) CALLSUB(52) REPSTART(53) REPEND(54) JUMP(55) TECHINFO(58) WEDMConditions(56) WEDMPassItem(57)	CLD[1]	CLD.SubCode	Postprocessor function type
a, b, c, d...	CLD[2] – CLD[257]		List of optional parameters. The structure of the parameters list is

Parameter	CLD array	CLD name	Description
			defined by the actual postprocessor function

The list of postprocessor functions types includes:

- PPFUN_PPFUN(500) - tool durability control command
- PPFUN_STARTSUB(50) - NC-subprogram start command
- PPFUN_ENDSUB(51) - NC-subprogram end command
- PPFUN_CALLSUB(52) - NC-subprogram call command
- PPFUN_TECHINFO(58) - command specifies technology parameters of the operation
- PPFUN_ENDTECHINFO(59) - end of operation's technological information
- PPFUN_WEDMConditions(56) - command specifies ED machining conditions
- Wire EDM pass properties PPFun WEDMPassItem(57)

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPFun
  case CLD.SubCode of
    50: begin ! StartSub
      OutBlock;
      Output ""
      Output "%"
      ProgNum = MainProgramID + CLD[2]; ProgNum@ = MaxReal
      if CLData$<>"" then
        Output OutStr$ + " (" + UPCASE(CLData$) + ")"
      OutBlock
      Interp_ = MaxReal; Interp_@ = Interp_
      Feed_ = MaxReal; Feed_@ = Feed_
    end
    51: begin ! EndSub
      OutBlock
      MSub = 99; MSub@ = MaxReal
      OutBlock
      Output "%"
    end
    52: begin ! CallSub
      OutBlock
      MSub = 98; MSub@ = MaxReal
      P = MainProgramID + CLD[2]; P@ = MaxReal
      OutBlock
    end
    58: begin ! TechInfo
      case cld[60] of
```

```
        1: OperationIsLathe = 0 !- mill,
        2: OperationIsLathe = 1 !- lathe,
        else OperationIsLathe = 3 ! other
      end
      CSOnCount = 0
      NeedKorDl = 0
      WasLoadTool = 0
      KorDl = 0; KorDl@ = KorDl
      H = cld[31]; H@ = H
      Zcycle = 0; Zcycle@ = Zcycle
      ZClear = 0; ZClear@ = ZClear
      QStep = 0; QStep@ = QStep
      !MC = 9; MC@ = Mc;
      Interp_ = MaxReal; Interp_@ = Interp_
      FEED_ = MaxReal; FEED_@ = FEED_
    end;
    59: begin ! EndTechInfo
      if Cmd.Int["PPFun(EndTechInfo).Enabled"]>0 then begin
        if CLDFile.CurrentFile <> CLDFile.FileCount - 1 then begin
          Output "M1"
        end
        Output " "
      end
      call CheckSwitchOffLocalCS
    end
  end
end
```

Access from .NET

Besides the base `OnPPFun`, the framework raises sub-code-specific handlers:

```
public override void OnPPFun(ICLDPPFunCommand cmd, CLDArray cld)
// base
public override void OnStartTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld) // TECHINFO(58)
public override void OnFinishTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld)// ENDTECHINFO(59)
public override void OnStartNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
// STARTSUB(50)
public override void OnFinishNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
// ENDSUB(51)
public override void OnCallNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
// CALLSUB(52)
```

Inheritance: `ICLDPPFunCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.SubCode`	int	Numerical code of the PPFUN subtype. Standard subtypes are listed above, but the code can have any arbitrary value defined by the postprocessor developer.
`cmd.TechInfo`	`IPPFunTechInfo`	For `TECHINFO(58)`/`ENDTECHINFO(59)` - additional information about the operation.

`IPPFunTechInfo` exposes `Enabled` (true if the operation uses a tool different from the previous one), `SetupStage` (`Index`, `Name`), `Tool` (`Number`, `ConnectorIndex`, `RevolverID`), `Workpiece` (`ConnectorID`) and `Part` (`Index`, `Number`, `Name`).

Example - `OnPPFun` (from the AgieCharmilles_WireEDM_DN postprocessor):

```
public override void OnPPFun(ICLDPPFunCommand cmd, CLDArray cld)
{
    switch (cmd.SubCode)
    {
        case 58: // TechInfo
            nc.WriteLine("(Rapid level       = " + Str((double)cld[9]) + ")");
            nc.WriteLine("(Upper guide level = " + Str((double)cld[47]) + ")");
            nc.WriteLine("(Upper work level  = " + Str((double)cld[10]) + ")");
            nc.WriteLine("(Lower work level  = " + Str((double)cld[11]) + ")");
            nc.WriteLine("(Lower guide level = " + Str((double)cld[48]) + ")");
            nc.WriteLine("(Wire diameter     = " + Str((double)cld[27]) + ")");
            break;

        case 56: // WEDMConditions
            nc.H.Show(cld[4]); // Offset code
            nc.HValue.Show(cld[5]); // Offset value
            nc.Block.Out();
            break;
```

```
        case 50: // STARTSUB
            nc.BlockN.Show(nc.ProgN + cld[2]);
            nc.Block.Out();
            break;

        case 51: // ENDSUB
            if (wireInserted)
            {
                BreakWire();
            }
            nc.MSub.Show(99);
            nc.Block.Out();
            nc.WriteLine();
            break;

        case 52: // CALLSUB
            nc.MSub.Show(98);
            nc.SubN.Show(nc.ProgN + cld[2]);
            nc.Block.Out();
            // Idle sub call
            // TODO: NCSub.Output(cld[2], 0)
            break;
    }
}
```

See also

- CLData access model

PPFUN CALLSUB(52) - NC-subprogram call command

NC-subprogram call command `PPFUN CALLSUB(52)` is used when NC-program contains several different NC-subprograms and at some point of program execution it is necessary to pass control to other subprogram. To achieve this the postprocessor of the `PPFUN CALLSUB(52)` command should call one of the NC-subprogram output operators. When this operators are called some or all of the NC-subprogram commands are processed. Otherwise technology commands of NC-subprograms are ignored.

`CLD[2]` parameter holds the unique identifier of the NC-subprogram.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `CALLSUB` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPFun
  case cld[1] of
      52: begin ! CallSub
        OutBlock
        MSub = 98; MSub@ = MaxReal
        P = MainProgramID + CLD[2]; P@ = MaxReal
        OutBlock
      end
  end
end
```

Access from .NET

In addition to the base `OnPPFun`, this sub-code raises the dedicated handler `OnCallNCSub`:

```
public override void OnCallNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
```

The common .NET model (members, 1-based vs 0-based index) is described in Postprocessor function (PPFUN).

Example - `OnCallNCSub` (from the Heidenhain_Mill_iTNC530_DN postprocessor):

```
public override void OnCallNCSub(ICLDSub cldSub, ICLDPPFunCommand cmd, CLDArray cld)
{
    // 25 ;Top plane
    // 26 CALL LBL101
    cldSub.Tag = 100 + cldSub.SubCode;
    nc.OutText($"CALL LBL{cldSub.Tag}");

    // Reset the state of main registers to output them at the first assign.
    nc.Block.Reset(nc.X, nc.Y, nc.Z, nc.Feed, nc.MoveType, nc.RCompens, nc.S,
nc.MCoolant, nc.MSpindle);

    // Idle run of cldSub translation (with disabled writing to the nc-file) to fill
registers with an actual state.
    NCFiles.DisableOutput();
    cldSub.Translate(false);
    NCFiles.EnableOutput();
}
```

See also

- Postprocessor function (PPFUN)
- CLData access model

PPFUN ENDSUB(51) - NC-subprogram end command

NC-subprogram end command `PPFUN ENDSUB(51)` together with the NC-subprogram start command `PPFUN STARTSUB(50)` is used to mark out the technology commands of the subprogram. All technology commands between PPFUN STARTSUB(50) and `PPFUN ENDSUB(51)`, are considered to be NC-subprogram commands. Main program commands are processed consecutively, whereas the NC-subprogram commands are skipped. The NC-subprogram commands are translated only when NC-subprogram output operators are called from one of the main program commands.

`CLD[2]` parameter holds the unique identifier of the NC-subprogram.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `ENDSUB` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPFun
  case cld[1] of
      51: begin ! EndSub
        OutBlock
        MSub = 99; MSub@ = MaxReal
        OutBlock
        Output "%"
      end
  end
end
```

Access from .NET

In addition to the base `OnPPFun`, this sub-code raises the dedicated handler `OnFinishNCSub`:

```
public override void OnFinishNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
```

The common .NET model (members, 1-based vs 0-based index) is described in Postprocessor function (PPFUN).

Example - `OnFinishNCSub` (from the Heidenhain_Mill_iTNC530_DN postprocessor):

```
public override void OnFinishNCSub(ICLDSub cldSub, ICLDPPFunCommand cmd,
CLDArray cld)
{
    // 454 LBL0
    nc.Block.Out();
    nc.OutText("LBL0");
}
```

See also

- Postprocessor function (PPFUN)
- CLData access model
- `PPFUN_STARTSUB(50)`

PPFUN ENDTECHINFO(59) - End of operation's technological information

This command is usually the last in the list of technological commands of the operation and serves to detect the end of the operation. The command has a set of parameters completely coincide with the parameters of PPFUN TECHINFO(58) command. It provides technological information on the operation of CAM system.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `ENDTECHINFO` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPFun
  case cld[1] of
      59: begin ! EndTechInfo
        if Cmd.Int["PPFun(EndTechInfo).Enabled"]>0 then begin
          if CLDFile.CurrentFile <> CLDFile.FileCount - 1 then begin
            Output "M1"
          end
          Output " "
        end
        call CheckSwitchOffLocalCS
      end
  end
end
```

Access from .NET

In addition to the base `OnPPFun`, this sub-code raises the dedicated handler `OnFinishTechOperation`:

```
public override void OnFinishTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld)
```

The common .NET model (members, 1-based vs 0-based index) is described in Postprocessor function (PPFUN).

Example - `OnFinishTechOperation` (from the Sinumerik (840D)_Mill_DN postprocessor):

```
public override void OnFinishTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld)
{
    if (cmd.TechInfo.Enabled) {
        if (cmd.CLDFile.Index != CLDProject.CLDFiles.FileCount - 1){
            nc.M.Show(1);
            nc.Block.Out();
        }
        nc.WriteLine();
    }
}
```

See also

- Postprocessor function (PPFUN)
- CLData access model
- `PPFUN TECHINFO(58)`

PPFUN PPFUN(500) - Tool durability control command

`PPFUN(500)` command is generated then is is required to change the tool because of it's wear. In that case the `CLD[2]` parameter's value is the tool durability represented in minutes.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `PPFUN` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example:

```
program PPFun
  case cld[1] of
    500: begin        ! PPFUN
      ! cld[2] holds the tool durability in minutes
    end
  end
end
```

Access from .NET

This sub-code has no dedicated handler - it is delivered through the base `OnPPFun` (dispatch on `cmd.SubCode`). See Postprocessor function (PPFUN).

See also

- Postprocessor function (PPFUN)
- CLData access model

PPFUN STARTSUB(50) - NC-subprogram start command

NC-subprogram start command `PPFUN STARTSUB(50)` together with the NC-subprogram end command `PPFUN ENDSUB(51)` is used to mark out the technology commands of the subprogram. All technology commands between `PPFUN STARTSUB(50)` and PPFUN ENDSUB(51), are considered to be NC-subprogram commands. Main program commands are processed consecutively, whereas the NC-subprogram commands are skipped. The NC-subprogram commands are translated only when NC-subprogram output operators are called from one of the main program commands.

`CLD[2]` parameter holds the unique identifier of the NC-subprogram.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `STARTSUB` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPFun
  case cld[1] of
      50: begin ! StartSub
        OutBlock;
        Output ""
        Output "%"
        ProgNum = MainProgramID + CLD[2]; ProgNum@ = MaxReal
        if CLData$<>"" then
          Output OutStr$ + " (" + UPCASE(CLData$) + ")"
        OutBlock
        Interp_ = MaxReal; Interp_@ = Interp_
        Feed_ = MaxReal; Feed_@ = Feed_
      end
  end
end
```

Access from .NET

In addition to the base `OnPPFun`, this sub-code raises the dedicated handler `OnStartNCSub`:

```
public override void OnStartNCSub(ICLDSub sub, ICLDPPFunCommand cmd, CLDArray cld)
```

The common .NET model (members, 1-based vs 0-based index) is described in Postprocessor function (PPFUN).

Example - `OnStartNCSub` (from the Heidenhain_Mill_iTNC530_DN postprocessor):

```
public override void OnStartNCSub(ICLDSub cldSub, ICLDPPFunCommand cmd,
CLDArray cld)
{
    //433 LBL101
    cldSub.Tag = 100 + cldSub.SubCode;
    nc.WriteLine();
    nc.OutText($"LBL{cldSub.Tag}");
}
```

See also

- Postprocessor function (PPFUN)
- CLData access model
- `PPFUN ENDSUB(51)`

PPFUN TECHINFO(58) - Technology parameters command

Technology parameters command `PPFUN TECHINFO(58)` passes additional technology information about the CAM system operation. `PPFUN TECHINFO(58)` command is generated at the beginning of technology commands list of an operation.

Parameters

CLD array	Parameter	Variable description	Group
1	SubCode	58 (TechInfo)	
2	2	CLData version	
3	3	Minimum X coordinate value	Toolpath box
4	4	Minimum Y coordinate value	
5	5	Minimum Z coordinate value	
6	6	Maximum X coordinate value	
7	7	Maximum Y coordinate value	
8	8	Maximum Z coordinate value	
9	9	Safe plane level (the level of rapid movement for the EDM operations)	Levels
10	10	Upper machining level	
11	11	Lower machining level	
12	12	Stock	Workpiece

CLD array	Parameter	Variable description	Group
13	13	Minimum X coordinate value	
14	14	Minimum Y coordinate value	
15	15	Minimum Z coordinate value	
16	16	Maximum X coordinate value	
17	17	Maximum Y coordinate value	
18	18	Maximum Z coordinate value	
19	19	CLData tolerance (number of decimal digits)	Tolerance
20	20	Measurement units: 0 – mm; 1 – inches.	
21	21	Minimum arc length	
22	22	Deviation off workpiece	
23	23	Deviation into workpiece	
24	24	Operation stock	
25	25	Tool type: 0 – cylindrical 1 – ball 2 – torus 3 – Double radial 4 – Limited double radial 5 – Conical mill 6 – Limited conical mill 7 – Engraver 8 – Drill 9 – Composite tool 10 –	Tool

CLD array	Parameter	Variable description	Group
		Mill with negative radius 11 – Cutting tool 12 – Tap 13 – Center drill 14 – Countersink drill 15 – Spot facing drill 16 – Reamer 17 – Threading drill 100 – External tool 101 – Boring tool 102 – External threading tool 103 – Internal threading tool 104 – External grooving tool 105 – Internal grooving tool 106 – Face grooving tool	
26	26	Tool number	
27	27	Tool diameter (wire diameter for the EDM operations)	
28	28	Work part length	
29	29	Work part angle (conical angle in radians)	
30	30	Round radius	
31	31	Length compensation number. Set to 0 if length compensation is off	
32	32	Radial compensation number. Set to 0 if radial compensation is off	

CLD array	Parameter	Variable description	Group
33	33	Length compensation value	
34	34	Radial compensation value	
35	35	Axial tool tooling point: 0 – Center, 1 – Custom, 2 – End. Lathe tool tooling point: 0 – Left, 1 – Right, 2 – Top, 3 – Bottom, 4 – Top left, 5 – Top right, 6 – Bottom left, 7 – Bottom right, 8 – Center, 9 – Custom.	
36	36	Spindle rotation (rev per min)	Speed
37	37	Rapid feedrate (mm per min or inch per min)	
38	38	Work feedrate	
39	39	Cut-in feedrate	
40	40	Approach feedrate	
41	41	Retraction feedrate	
42	42	Next feedrate	
43	43	Return feedrate	
44	44	Toolpath length	Toolpath properties
45	45	Operation's machining time in minutes	
46	46	Total time of the techprocess in minutes	

CLD array	Parameter	Variable description	Group
47	47	Upper guide level of the EDM machine	Levels
48	48	Lower guide level of the EDM machine	
49	49	Operation's working time in minutes	Toolpath properties
51	51	Tool Axis: 1 – X, 2 – Y, 3 – Z. (Lathe – 2, Lathe-drill – 1, Milling – 3)	
52	52	X coordinate	Tool change point
53	53	Y coordinate	
54	54	Z coordinate	
55	55	X coordinate	Reference point
56	56	Y coordinate	
57	57	Z coordinate	
58	58	Active tube number: 0 – coolant off, 1 – flood, 2 – mist, 3 – tool.	
60	60	Operation technology group: 0 – not set, 1 – milling, 2 – lathe, 3 – auxiliary, 4 – electric discharge.	
61	61	0 – PPFUN command is not activated 1 – PPFUN command is activated	

CLD array	Parameter	Variable description	Group
62	62	Robot kinematics: 0 – the position of the 3rd joint doesn't depend on the position of the 2nd joint 1 – the 2nd and 3rd joints are linked with the parallelogram mechanism	Robot kinematics
63	63	Part number that is explicitly set by the user in the part parameters of CAM system	Part
64	64	Turn tool tool tip offset direction vector code (0-8): 0: X=0, Y=0; 1: X=+1, Y=+1; 2: X=-1, Y=+1; 3: X=-1, Y=-1; 4: X=+1, Y=-1; 5: X=+1, Y=0; 6: X=0, Y=+1; 7: X=-1, Y=0; 8: X=0, Y=-1;	Tool tip code
65	65	Workpiece coordinate system number (G54, G55, G54.2 etc) of the operation	WCS number
80	80	Step along tool axis for multilayer machining (For 2D Contour operation only)	Layers
81	81	Finish layer stock along tool axis (For 2D Contour operation only)	Layers

Parameters available through the cmd operator

Type	Description	Subfield description
TCLDPPFun: ComplexType	The command determining technological parameters of the operation	
PPFun: Array, Key="RecordID"	cmd.Ptr["PPFun"] - An array of structures of different types to storage parameters of the PPFun command	
TPPFunTechInfoParameters: ComplexType	cmd.Ptr["PPFun"].Item[1] or cmd.Ptr["PPFun(TechInfo)"] - Separate element of the PPFun array. Contains information on technological parameters of the operation. Access to the array elements can either index or by key field RecordID. For the command of determining of operation technological parameters it always has the value "TechInfo".	
RecordID: String	cmd.Str["PPFun(TechInfo).RecordID"] - Postprocessor function type's identifier. In this case always has a value "TechInfo".	
RecordCode: Integer	cmd.Int["PPFun(TechInfo).RecordCode"] - The code that identifies the type of postprocessor function. In this case always has a value "58".	
Operation: Array	cmd.Ptr["PPFun(TechInfo).Operation"] - almost a full list of parameters of technological operation. As the child elements of the item are a variety of operation parameters: the type of operation, its name, group,	

Type	Description	Subfield description
	tool properties, stocks, calculation tolerances, lead-in and lead-out methods, machining levels and step, etc. Specific set of parameters depends on the type of operation. Type of operation can be obtained as cmd.Ptr["PPFun(TechInfo).Operation(1)"].Name.	
SetupStage: ComplexType	cmd.Ptr["PPFun(TechInfo).SetupStage"] - Information about the Setup stage in which the operation is located.	
	Index: Integer	Index of the setup stage in the list of CAM system's project setup stages
	Name: String	The name of the setup stage group inside CAM system's project
Tool: ComplexType	cmd.Ptr["PPFun(TechInfo).Tool"] - Parameters relating to the tool used in technological operation.	
	ID: Integer	cmd.Int["PPFun(TechInfo).Tool.ID"] - tool identifier (number)
	HolderID: Integer	cmd.Int["PPFun(TechInfo).Tool.HolderID"] - tool holder identifier (number)
	RevolverID: String	cmd.Str["PPFun(TechInfo).Tool.RevolverID"] - String identifier of the turret head.
Workpiece: ComplexType	cmd.Ptr["PPFun(TechInfo).Workpiece"] - Parameters relating to	

Type	Description	Subfield description
	the workpiece used in technological operation.	
	HolderID: String	cmd.Str["PPFun(TechInfo).Workpiece.HolderID"] - string identifier of the workpiece connector.
Part: ComplexType	cmd.Ptr["PPFun(TechInfo).Part"] - Information about the Part being machined by this operation	
	Index: Integer	Index of the Part in the list of CAM system's project parts list
	Number: Integer	Part number that is explicitly set by the user in the part parameters of CAM system
	Name: String	The Part name (group name) that is set in CAM system

Below is an example of access to the properties of operation by the operator cmd.

```
program PPFun
  if cld[1]=58 then begin ! TechInfo
    call AnalysePPFun
  end
end
sub AnalysePPFun
  OpType: String
  OpGroup: Integer
  ! String name of operation type: TSTRoughingWaterlineOp, TST2DContouringOp,
TSTDrillOp, LatheRoughOp...
  OpType = cmd.Ptr["PPFun(TechInfo).Operation(1)"].Name
  ! OperationGroup:
  ! 1 - Milling
  ! 2 - Lathe
  ! 3 - Auxiliary
  ! 4 - WireEDM
  OpGroup = cmd.Int["PPFun(TechInfo).Operation(1).OperationGroup"]
  Output "; Operation type = " + OpType
  Output "; Operation comment = " + cmd.Str["PPFun(TechInfo).Operation(1).Comment"]
```

```
  if OpGroup=1 then begin ! Milling
    Output "; Tool overhang = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).ToolSection.Tools(1).AxialOverhang"])
    if OpType="TSTRoughingWaterlineOp" then begin
      Output "; Axial stock = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).Stocks.Axial"])
      Output "; Radial stock = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).Stocks.Radial"])
    end
  end else if OpGroup=2 then begin ! Lathe
    Output "; Tool X overhang = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).ToolSection.Tools(1).Overhang.Value.X"])
    Output "; Tool Y overhang = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).ToolSection.Tools(1).Overhang.Value.Y"])
    Output "; Tool Z overhang = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).ToolSection.Tools(1).Overhang.Value.Z"])
    Output "; Axial stock = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).LatheStock.AxialValue"])
    Output "; Radial stock = " +
Str(cmd.Flt["PPFun(TechInfo).Operation(1).LatheStock.RadialValue"])
  end
subend
```

Access from .NET

In addition to the base `OnPPFun`, this sub-code raises the dedicated handler `OnStartTechOperation`:

```
public override void OnStartTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld)
```

The common .NET model (members, 1-based vs 0-based index) is described in Postprocessor function (PPFUN).

Reading the operation's XML properties by name

For `TECHINFO`/`ENDTECHINFO`, besides the fixed parameters above, the command bridges into the operation's XML properties by name through the `cmd` helpers:

> Warning. This is a direct read of the CAM system's internal data structures. Their layout and naming are not guaranteed to stay the same between versions. Use it only as a last resort, when the value you need is not exposed through the public CLData API above. Prefer the documented parameters and typed members whenever they cover your case.

The array element index differs between environments - 1-based in sppx, 0-based in .NET:

- sppx: `cmd.Ptr["PPFun(TechInfo).Operation(1)"]`
- .NET: `cmd.Ptr["PPFun(TechInfo).Operation(0)"]`

Example (from the Heidenhain (iTNC530)_Mill postprocessor - sppx, 1-based element index):

```
! a deep operation property, reached through the operation's TechInfo command
MinDepth =
cmd.Flt["PPFun(TechInfo).Operation(1).ChipBreaking.DepthDegression.MinStepPercent"]
```

To find the exact name of a property: open the operation's property inspector, locate the property, click the ... button at the right of its row and choose "Copy property name to clipboard"; or inspect the operation's XML. The naming rules and the full access model are described in Using XML properties from code.

Example - `OnStartTechOperation` (from the Heidenhain_Mill_iTNC530_DN postprocessor):

```
public override void OnStartTechOperation(ICLDTechOperation op, ICLDPPFunCommand
cmd, CLDArray cld)
{
    nc.WriteLine();
    nc.OutText("; " + Transliterate(op.Comment));

    // 17 TOOL CALL 1 Z S159 DL+0 DR+0
    // 18 TOOL DEF 2
    // 19 L Z+0 FMAX M91
    if (op.Tool.Command!=null && op.Tool.Command.ToolChanged) { // if the operation
has a Load tool command
        double s = 0;
        if (op.SpindleCommand!=null)
            s = Abs(op.SpindleCommand.RPMValue);
        nc.OutText($"TOOL CALL {op.Tool.Number} {GetToolPlaneName(op.Tool.Command)}
{nc.S.ToString(s)} DL+0 DR+0");

        int nextTool = FindNextOpToolNumber(op);
        if (nextTool>0)
            nc.OutText($"TOOL DEF {nextTool}");

        if (firstToolNumber<0)
            firstToolNumber = op.Tool.Number;
        nc.OutText("L Z+0 FMAX M91");
    }
```

```
}
```

See also

- Postprocessor function (PPFUN)
- CLData access model
- `PPFUN_ENDTECHINFO(59)`
- Using XML properties from code

PPFUN WEDMConditions(56) - EDM machining conditions specification command

EDM machining conditions specification command `PPFUN WEDMConditions(56)` is generated by EDM operations of CAM system. This command is used to pass EDM operation's machining conditions table to the postprocessor. Each row in the machining conditions table represents machining conditions of a single pass. So thereafter the number of rows in the table is equal to the number of passes used. Each table row is passed by a separate `PPFUN WEDMConditions(56)` command. So the list of technology commands of every EDM operation is opened by the list of `WEDMConditions(56)` commands, the number of commands is equal to the number of rows in the EDM operations conditions table.

Parameters

Parameters	CLD array	CLD name	Description
Command SubCode	CLD[1]	CLD.SubCode	WEDMCONDITIONS(56) – a row of machining conditions table of an EDM operation
Pass ordinal number	CLD[2]		Run's ordinal numbers are usually assigned sequentially starting with 1. However, ordinal numbers can start with number other than 1, depending on operation settings in CAM system
Pass condition code	CLD[3]		Condition code specifies machining conditions for the given pass. Condition code changing in the course of the program is done by FEDRAT command.
Offset Code	CLD[4]		Offset code defines the corrector number which should be used to

Parameters	CLD array	CLD name	Description
			specify correction value for the given pass. Corrector number switching is done by CUTCOM command.
Correction value	CLD[5]		Correction value for the given corrector number
Feed value	CLD[6]		Wire feed value in mm per min (inches per min)
Additional parameters	CLD[100] – CLD[256]		List of additional parameters for each pass. The number of additional parameters is specific for each postprocessor.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `WEDMConditions` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example:

```
program PPFun
  case cld[1] of
    56: begin        ! WEDMConditions
      ! a row of the EDM machining-conditions table
    end
  end
end
```

Access from .NET

This sub-code has no dedicated handler - it is delivered through the base `OnPPFun` (dispatch on `cmd.SubCode`). See Postprocessor function (PPFUN).

See also

- Postprocessor function (PPFUN)
- CLData access model

PPFUN WEDMPassItem(57) - Wire EDM pass properties

Wire EDM pass properties command WEDMPassItem(57) is generated by EDM operations of CAM system. It is used to transfer information to the postprocessor about which contour and pass of toolpath the nested CLData commands belong to. It appears at the start of contour movements and before the end of contour movements for each separate pass.

Parameters

Parameters	CLD array	CLD name	Description
Command SubCode	CLD[1]	CLD.SubCode	WEDMPassItem(57) – the type of PPFun command - Wire EDM pass properties.
On/Off	CLD[2]		Contour start-stop flag: On(71) - start of contour, Off(72) - end of contour.
Contour type	CLD[3]		The type of contour - general type pass (0) or coreless pocketing pass (1).
Item index	CLD[4]		Index of this contour in the list of Job assignment items of the operation.
Pass index	CLD[5]		Offset pass index
Pass type	CLD[6]		The type of pass: 0 - Rough, 1 - Rough with bridge, 2 - bridge, 3 - finish together, 4 - finish separate.

Access from sppx

PPFUN has no sub-code-specific handler in sppx. All PPFUN sub-codes are processed in the single `program PPFun` handler, dispatching on `cld[1]` (`CLD.SubCode`); handle `WEDMPassItem` by adding a branch there. See Postprocessor function (PPFUN) for the full handler.

Example:

```
program PPFun
  case cld[1] of
    57: begin        ! WEDMPassItem
      ! wire-EDM pass properties
    end
  end
end
```

Access from .NET

This sub-code has no dedicated handler - it is delivered through the base `OnPPFun` (dispatch on `cmd.SubCode`). See Postprocessor function (PPFUN).

See also

- Postprocessor function (PPFUN)
- CLData access model

PPRINT - Postprocessor printing

Command caption

How the command appears in the CLData command list:

```
PPRINT "..."
```

PPRINT command passes text information from CAM system to postprocessor. The text of technology command is put into the CLDATA$ predefined variable, after that the text can be printed out into the operator window as a comment explaining postprocessor operation.

Toolpath tags

Besides free-form text, PPRINT is also used to tag key points of the toolpath. Such a tag is a PPRINT whose text starts with `#`, followed by the tag type and, optionally, a value:

```
PPRINT "#KeyPoint: StartCutting"
PPRINT "#Approach:[Rule 1]=C; Z10; X; Z"
PPRINT "#Return:[Rule 1]=Z10; X; Z; C"
```

The CAM system inserts these tags to annotate the toolpath - for example the start of cutting, and the rule that produced the approach or return move. A postprocessor can recognize a tag by the leading `#`, react to it if needed, or simply skip it so the marker does not reach the NC program. See The structure of a CLData listing for where these tags appear inside a real operation.

Access from sppx

Handler: `program PPrint`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program PPrint
  print cldata$
end
```

Access from .NET

Handler:

```
public override void OnPrint(ICLDPrintCommand cmd, CLDArray cld)
```

Inheritance: `ICLDPrintCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.Text`	`string`	Arbitrary text information transmitted from CAM.

Example:

```
public override void OnPrint(ICLDPrintCommand cmd, CLDArray cld)
{
    nc.OutWithN("(" + cmd.Text + ")");   // pass the CAM text into the NC program
(here - as a comment)
}
```

See also

- CLData access model

RAPID - Rapid feedrate

Command caption

How the command appears in the CLData command list:

```
RAPID N n
```

RAPID activates rapid travel mode (G-code G00). All further travels are done in rapid mode until feedrate modification command FEDRAT is received.

Access from sppx

Handler: `program Rapid`

Parameter	CLD array	CLD name	Description
n	CLD[1]	CLD.N	Rapid feedrate value

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Rapid
  if IsRotaryInterpolation>0 then begin
    INTERP_ = 1
    Feed_ = 5000
    GFeed = 94
  end else begin
    if INTERP_ > 0 then INTERP_ = 0    ! G0
  end
  RapFr=cld[1]
end
```

Access from .NET

Handlers:

```
public override void OnMoveVelocity(ICLDMoveVelocityCommand cmd, CLDArray cld)   //
shared velocity handler (FEDRAT and RAPID)
public override void OnRapid(ICLDRapidCommand cmd, CLDArray cld)   // rapid feed
```

Inheritance: `ICLDRapidCommand : ICLDMoveVelocityCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsRapid`	`bool`	Contains "true" if the command is a RAPID command, otherwise it is "false".
`cmd.FeedKind`	`CLDFeedKind`	The kind of a feed (working, rapid, engage, return, finish etc.). Possible values: `Working` (0) - The feed kind for base working movements. `Rapid` (1) - The feed kind of rapid movements. `First` (2) - The feed kind for the working movements on a first (roughing) pass. `Engage` (4) - The feed kind for the movements entering to the first machining (cutting) point. `Retract` (8) - The feed kind for the movements leaving the last machining (cutting) point. `Plunge` (16) - The feed kind for the movements that plunge vertically into the machining area. `Finish` (32) - The feed kind for the working movements on a last (finishing) pass. `Next` (64) - The feed kind for the short link movements. `Return` (128) - The feed kind for the movements which take the tool away from the machining site.

Member	Type	Description
		`Approach` (256) - The feed kind for the movements which bring the tool closer to the machining site. `Rapid5D` (512) - The feed kind value reserved for an internal use. `TransitionOnSafe` (1024) - The feed kind for the movements performed in a safe area. `ApproachFromSafe` (2048) - The feed kind for the movements from the safe area to the machining side. `ReturnToSafe` (4096) - The feed kind for the movements from the machining side to the safe area. `LongNext` (8192) - The feed kind for the long link movements. `ThreadPitch` (16384) - The feed kind for the threading movements. `Overload` (1073741824) - The feed kind value reserved for an internal use.
`cmd.FeedCode`	`int`	Numerical representation of the "FeedKind" flag, that defines the kind of a feed (working, rapid, engage, return, finish etc.). For a wire EDM case it contains the numerical code of a row in a machining conditions table.
`cmd.FeedUnits`	`CLDFeedUnits`	Units of the feed (mm/min, mm/rev etc.) Possible values: `MPM` (0) - Current linear units

Member	Type	Description
		(mm or inch) per minute. `MPR` (1) - Current linear units (mm or inch) per revolution. `ConditionCode` (2) - Numerical code of a row in a machining conditions table (for a wire EDM).
`cmd.FeedValue`	`double`	The value of the feed.

Example - `OnMoveVelocity` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnMoveVelocity(ICLDMoveVelocityCommand cmd, CLDArray cld)
{
    if (cmd.IsRapid) {
        nc.GInterp.v = 0;
    } else {
        if (nc.GInterp == 0)
            nc.GInterp.v = 1;
        nc.F.v = cmd.FeedValue;
        if (cmd.FeedUnits==CLDFeedUnits.MPR)
            nc.GFeed.v = 95;
        else
            nc.GFeed.v = 94;
        if (IsFirstFeed)
        {
            nc.F.Show();
            nc.GFeed.Show();
            IsFirstFeed = false;
        }
    }
}
```

See also

- CLData access model

ROTABL - Table rotation

> Deprecated. This command is obsolete - the current version of the CAM system does not generate it automatically. It is documented here for postprocessors that still handle legacy CLData.

Command caption

How the command appears in the CLData command list:

```
ROTABL ABS(0), A ab, CCLW(59)|CLW(60), PLANE XY(33)|YZ(37)|XZ(41)
```

ROTABL command passes positioning of the rotatable axis of the machine. This command is obsolete and is supported for backwards compatibility. Process the MULTIGOTO command to handle rotatable axes.

Access from sppx

Handler: `program Rotabl`

or

```
ROTABL INCR(66), B ab, CCLW(59)|CLW(60), PLANE XY(33)|YZ(37)|XZ(41)
```

Parameter	CLD array	CLD name	Description
ABS(0) INCR(66)	CLD[1]	CLD.Incr	Rotation type: 0 (ABS) absolute, 66 (INCR) incremental
ab	CLD[2]	CLD.AB	Table rotation angle value or increment value
CCLW(59) CLW(60)	CLD[3]	CLD.CCLW	Rotation direction: CCLW – counter clockwise; CLW – clockwise.

Parameter	CLD array	CLD name	Description
XY(33) YZ(37) XZ(41)	CLD[4]	CLD.Plane	Plane

Use ISO.A, ISO.B, ISO.C to create similar ISO postprocessors with masks. The system fills these variables automatically with values of CLD array items:

CLD parameters value	ISO value
CLD[4] = 33 CLD[1] = 0 CLD[3] = 59	ISO.C = CLD[2]
CLD[4] = 33 CLD[1] = 0 CLD[3] = 60	ISO.C = -CLD[2]
CLD[4] = 33 CLD[1] = 66	ISO.C = CLD[2]
CLD[4] = 41 CLD[1] = 0 CLD[3] = 59	ISO.B = CLD[2]
CLD[4] = 41 CLD[1] = 0 CLD[3] = 60	ISO.B = -CLD[2]
CLD[4] = 41 CLD[1] = 66	ISO.B = CLD[2]
CLD[4] = 37 CLD[1] = 0 CLD[3] = 59	ISO.A = CLD[2]
CLD[4] = 37 CLD[1] = 0 CLD[3] = 60	ISO.A = -CLD[2]
CLD[4] = 37 CLD[1] = 66	ISO.A = CLD[2]

Example (from the HAAS (SL30)_TurnMill postprocessor):

```
program RoTabl
!  using RotAng,RotAngStep,RotPlan initialize in PARTNO
!  if cld[3] = 59 then sing = 1 else sign = -1
!  if cld[1] = 0 then RotAng = cld[2]*sign
!  if cld[1] = 66 then RotAngStep = cld[2]*sign
!  if cld[4] = 33 then RotPlan = 17        ! C-table rotationn plane XY
!  else if cld[4] = 37 then RotPlan = 18   ! B-table rotationn plane XZ
!  else if cld[4] = 41 then RotPlan = 19   ! A-table rotationn plane YZ
end
```

See also

- CLData access model

SAFPOS - Tool change point

> Deprecated. This command is obsolete - the current version of the CAM system does not generate it automatically. It is documented here for postprocessors that still handle legacy CLData.

Command caption

How the command appears in the CLData command list:

```
SAFPOS X x, Y y, Z z, N n
```

SAFPOS command defines the coordinates of the point, where the tool center will be positioned for the change of tool (LOADTL command), are defined. Tool change make in the current point as default.

Access from sppx

Handler: `program SafPos`

Parameter	CLD array	CLD name	Description
x y z	CLD[1] CLD[2] CLD[3]	CLD.X CLD.Y CLD.Z	Coordinates of point of tool change
N	CLD[4]	CLD.N	The number of point of tool change in the NC-machine

See also

- CLData access model

SELWORKPIECE - Active workpiece connector selection

Command caption

How the command appears in the CLData command list:

```
SELWORKPIECE WorkpieceConnectorID
```

Use the SELWORKPIECE command to specify active workpiece connector. The workpiece connector is a machine unit that is used to fix the workpiece. For the multi-spindle lathe machines this command specifies the active spindle. For the milling machines the workpiece connector can be the pallet for example. All machining commands will relate to the workpiece fixed in the active connector.

Access from sppx

Handler: `program SelWorkpiece`

The SELWORKPIECE command's single parameter is the string identifier of the workpiece connector. This identifier is defined by the CAM system machine scheme. Parameter value is stored into the CLDATA$ predefined variable. Following is an example of a program processing this command.

```
program SelWorkpiece
  if CLData$="MainSpindle" then
    Output "SETMS(1)" ! Activate the main spindle
  else
    Output "SETMS(2)" ! Activate the counter spindle
end
```

Access from .NET

Handler:

```
public override void OnSelWorkpiece(ICLDSelWorkpieceCommand cmd, CLDArray cld)
```

Inheritance: `ICLDSelWorkpieceCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.ConnectorID`	`string`	Textual workpiece connector identifier (as it defined inside machine schema).

Example:

```
public override void OnSelWorkpiece(ICLDSelWorkpieceCommand cmd, CLDArray cld)
{
    activeWorkpieceConnector = cmd.ConnectorID;   // remember which workpiece
connector the following commands refer to
}
```

See also

- CLData access model

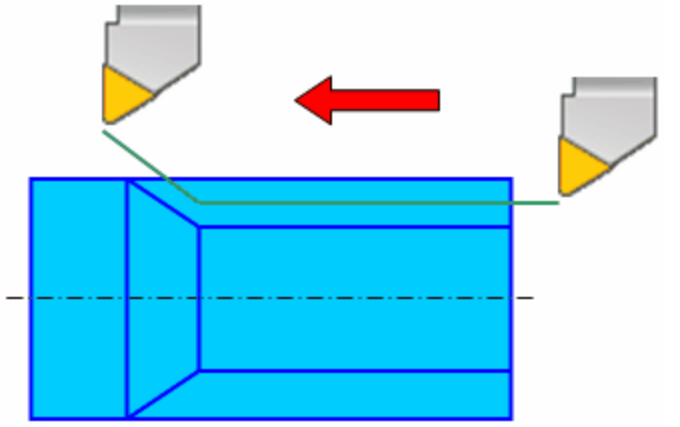
SINGLETHREAD - Single-pass thread

Command caption

How the command appears in the CLData command list:

```
SINGLETHREAD OD(0) | ID(1) | FACE(2), STEP(0) | COUNT(1), s, ANGLE a
```

SINGLETHREAD command is used in continuous cylindrical or conical thread cutting with constant step by a lathe tool. Commonly this is the G32 or G33 G-codes in NC-programs.



When the threading mode is activated the tool movement and spindle rotation synchronization mode is activated. All following movements of the tool are performed in this mode until feed rate change (FEDRAT) command or rapid travel (RAPID) command is received. If the tool is traveling along the spindle axis then cylindrical thread is formed. If the tool is traveling along the spindle axis and also perpendicular to it then conical thread is machined. It is possible to machine a face thread by moving the tool only perpendicularly to the spindle axis.

SINGLETHREAD command parameters define the threading orientation, the thread pitch and the spindle start angle.

There are two ways to specify the thread pitch: by specifying the value of the pitch, or by specifying the number of thread turns per length unit.

Spindle start angle is specified in degrees and defines the angle that is laid off the zero signal of the spindle probe before thread cutting. Specify different angle values for different passes to machine multi-start thread.

Access from sppx

Handler: `program SingleThread`

Parameter	CLD array	CLD name	Description
OD(0) ID(1) FACE(2)	CLD[1]	CLD.Orientation	Threading orientation: OD(0) – external, ID(1) – internal, FACE(2) – face.
STEP(0) COUNT(1)	CLD[2]	CLD.IsCount	Thread pitch defining: STEP(0) – value, COUNT(1) – number of thread turns per length unit.
s	CLD[3]	CLD.Value	Thread pitch value.
a	CLD[4]	CLD.StartAngle	Spindle start angle (for multiple-start thread).

Example (from the Fanuc (30i)_Mill postprocessor):

```
program SingleThread
  Feed_ = cld[3]; Feed_@ = MaxReal
  QStartAngle = cld[4]; QStartAngle@ = MaxReal
  Interp_ = 33; Interp_@ = MaxReal
end
```

Access from .NET

Handler:

```
public override void OnSinglePassThread(ICLDSinglePassThreadCommand cmd,
CLDArray cld)
```

Inheritance: `ICLDSinglePassThreadCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.Orientation`	`CLDThreadOrient`	Thread orientation (OD, ID, Face)
`cmd.IsODThread`	`bool`	Returns "True" if it's an external thread cutting command.
`cmd.IsIDThread`	`bool`	Returns "True" if it's an internal thread cutting command.
`cmd.IsFaceThread`	`bool`	Returns "True" if it's a face thread cutting command.
`cmd.Units`	`CLDThreadUnits`	Thread pitch units (mm or TPI).
`cmd.StepIsDistance`	`bool`	Returns "True" if thread pitch specified in a linear units (usually in mm).
`cmd.StepIsTPU`	`bool`	Returns "True" if thread pitch specified as a Threads per units (usually as threads per inch).
`cmd.Step`	`double`	Thread pitch value.
`cmd.StartAngle`	`double`	Thread start angle in degrees. Frequently can be used for a multi-start threads.

Example - `OnSinglePassThread` (from the Goodway SW postprocessor):

```
public override void OnSinglePassThread(ICLDSinglePassThreadCommand cmd,
CLDArray cld)
{
        nc.GInterp.v = 32;
        if (cmd.StepIsTPU)
            nc.F.v = 1.0/cmd.Step;
        else
            nc.F.v = cmd.Step;
        nc.QThreadAngle.v = cmd.StartAngle;
        nc.QThreadAngle.v0 = 0;
}
```

See also

- CLData access model

SPINDL - Spindle

Command caption

How the command appears in the CLData command list:

```
SPINDL ON(71)|OFF(72)|ORIENT(246), NO n, K k, MODE RPM(0)|RANGE(1)|CSS(2) {,
SPEED c}
```

SPINDL command passes spindle control parameters. SPINDL command is used to define spindle rotation frequency or range of frequencies, surface speed, etc.

Access from sppx

Handler: `program Spindl`

The `cld` array is read by index or by name - in sppx by the typed short name (e.g. `cld.X`), in .NET by the same name as a string key (e.g. `cld["X"]`).

Parameter	CLD array	CLD name	Description
ON(71) OFF(72) ORIENT(246)	CLD[1]	CLD.OnOff	Command type: ON(71) – spindle rotation on, OFF(72) – spindle rotation off, ORIENT(246) – oriented spindle stop.
n	CLD[2]	CLD.NO	Rotation frequency, spindle rotation angle or maximum spindle rotation frequency in CSS mode.
k	CLD[3]	CLD.K	Range of rotation frequencies.
RPM(0) RANGE(1) CSS(2)	CLD[4]	CLD.Mode	Mode of the rotation frequency definition: RPM(0) – revolutions per minute, RANGE(1) – range of spindle

Parameter	CLD array	CLD name	Description
			rotation frequencies, CSS(2) – constant surface speed mode.
c	CLD[5]	CLD.CSS	Constant surface speed value for the CSS mode.

To create ISO postprocessor using masks use ISO.M values which correspond to SPINDL parameters.

CLD parameter	ISO value
CLD[1] = 71	ISO.M = 3
CLD[1] = 72	ISO.M = 5

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Spindl
  OutBlock
  case cld[1] of
    71: begin ! On
      if cld[4]=2 then begin ! CSS
        GSpindl = 96
        S = abs(cld[5])
        OutBlock
        if cld[5] > 0 then SpDir = 3
                      else SpDir = 4    ! M3/M4  CW/CCW
        SpRPM = abs(cld[2])
      end else begin ! RPM
        if cld[2] > 0 then Msp = 3
                      else Msp = 4    ! M3/M4  CW/CCW
        if ((SpDir=3) or (SpDir=4)) and (SpDir<>Msp) then ! Spindle reverse
          Output "M5"
        GSpindl = 97
        S = abs(cld[2]); S@=SpRPM
        Msp@ = SpDir
        SpDir = Msp
        SpRPM = abs(cld[2])
        OutBlock
      end
    end
    72: begin ! Off
```

```
      Msp = 5
      S=0; S@=S
      SpRPM = 0
      SpDir = 5    ! M3/M4/M5
      OutBlock  ! Msp = 5;; OutBlock; Msp=1; OutBlock
    end
    246: begin  ! M19 ORIENT
      OutBlock
      M = 19; M@ = MaxReal
      OutBlock
    end
  end
end
```

Access by parameter name (helpers)

Parameters available through the `cmd` operator (the same in sppx and .NET):

Name	Helper	Description
`Action`	`cmd.Int["Action"]`	Command kind (0 - on, 1 - off, 2 - orient).
`RotationDirection`	`cmd.Int["RotationDirection"]`	Rotation direction.
`SpeedMode`	`cmd.Int["SpeedMode"]`	0 - RPM, 1 - RANGE, 2 - CSS.
`Value`	`cmd.Flt["Value"]`	Rotation frequency / orientation angle / max rpm.
`CSSValue`	`cmd.Flt["CSSValue"]`	Constant surface speed value.
`RangeValue`	`cmd.Int["RangeValue"]`	Gearbox range.
`SpindleID`	`cmd.Str["SpindleID"]`	Spindle identifier.

Access from .NET

Handler:

```
public override void OnSpindle(ICLDSpindleCommand cmd, CLDArray cld)
```

Inheritance: `ICLDSpindleCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd[...]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Property	Type	Description
`cmd.SpindleID`	string	Textual identifier of the spindle for which this command sets parameters.
`cmd.Action`	`CLDSpindleAction`	The action for the spindle (`On`, `Off` or `Orient`).

`cmd.IsOn`
`cmd.IsOff`
`cmd.IsOrient`
	bool	True if it is a switch-ON / switch-OFF / angular-orientation command.

`cmd.SpeedMode`	`CLDSpindleSpeedMode`	The spindle rotation mode (`RPM` or `CSS`).
`cmd.IsRPM` `cmd.IsCSS`	bool	True if the mode is RPM (rev/min, G97) / CSS (constant surface speed, G96).
`cmd.RotationDir`	`CLDRotationDir`	Rotation direction for the spindle (CW or CCW).
`cmd.IsClockwiseDir`	bool	True if the rotation direction is CW (clockwise).
`cmd.RPMValue`	double	Revolutions per minute (maximal if the CSS mode is active).
`cmd.CSSValue`	double	The value of a linear (surface) speed for the spindle.
`cmd.GearRange`	int	Spindle's gearbox range.

Example (from the Sinumerik 840D mill postprocessor):

```
public override void OnSpindle(ICLDSpindleCommand cmd, CLDArray cld)
{
```

```
    if (cmd.IsOn)
    {
        switch (cmd.SpeedMode)
        {
            case CLDSpindleSpeedMode.RPM:
                nc.S.Show(cld[2]);
                nc.Msp.Show(cmd.IsClockwiseDir ? 3 : 4);
                break;
            case CLDSpindleSpeedMode.CSS:
                throw new Exception("CSS mode not realized");
        }
    }
    else if (cmd.IsOff)
        nc.Msp.Show(5);
}
```

See also

- CLData access model

STOP - Stop

Command caption

How the command appears in the CLData command list:

```
STOP
```

Use the STOP command to form frames of breaking the execution of NC-program. Commonly it's the M00 code of NC-program. When the stop frame is reached execution of the NC-program is interrupted.

Access from sppx

Handler: `program Stop`

Example (from the Fanuc (30i)_Mill postprocessor):

```
program Stop
  if M <> M@ then OutBlock           ! output to NC block
  M = 0                                ! M0
  M@ = 1
end
```

Access from .NET

Handler:

```
public override void OnStop(ICLDStopCommand cmd, CLDArray cld)
```

Inheritance: `ICLDStopCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Example - `OnStop` (from the Fanuc (30i)_Mill_DN postprocessor):

```
public override void OnStop(ICLDStopCommand cmd, CLDArray cld)
{
    nc.Block.Out();
    nc.M.Show(0);
```

```
    nc.Block.Out();
}
```

See also

- CLData access model

STRUCTURE - Toolpath structure

Command caption

How the command appears in the CLData command list:

```
STRUCTURE ON(71)|OFF(72), TYPE(Level), Comment(Level: -65)
```

The STRUCTURE command allows us to represent a tree structure of CLData commands as a linear list of commands.

Access from sppx

Handler: `program Structure`

In CAM system toolpath of each operation is represented as a tree, separated into groups and subgroups, within which are placed the usual commands of moving tool, turn on the spindle or feed, etc. Inside the postprocessors generator toolpath is represented as a linear list of commands that is not divided into groups. STRUCTURE command intended to preserve the original information as to in what group was originally located CLData command. Each group is converted into two commands: `STRUCTURE ON(71)` and `STRUCTURE OFF(72)`. The first command defines the start of the group, and the second one - the end of the group. All items that are placed inside a group will be located between these two commands. As well as groups can be nested within each other, inside a `STRUCTURE ON(71)`... `STRUCTURE OFF(72)` block can be nested several other blocks `STRUCTURE ON(71)`... `STRUCTURE OFF(72)`.

Parameter	CLD array	Description
ON(71) OFF(72)	CLD[1]	Sign that this command is the beginning or the end of the group: ON(71) - command defines the beginning of group, OFF(72) - command defines the end of group.

Parameters available through the cmd operator

TCLDStructure: ComplexType	Command that defines the structure
IsClose: Integer	cmd.Int["IsClose"] - Sign that this command is the beginning or the end of the group: 0 - command defines the beginning of group, 1 - command defines the end of group.
NodeType: String	cmd.Str["NodeType"] - A string that specifies the type of group. For example, Approach - a group of approach movings, Level - a layer of machining, etc. This string is not translated into local languages.
Comment: String	cmd.Str["Comment"] - The group name that is displayed in the CAM system's toolpath tree. Can be translated into local languages and contain arbitrary characters.

Example (from the KUKA robot postprocessor):

```
program Structure
  str1, str2: string
  str1 = Cmd.Str["NodeType"]
!  str2 = Cmd.Str["Comment"]

  if CLD.OnOff=71 and drilling = 10 then begin
    cdis = ""
  end

  if CLD.OnOff=72  and drilling = 10 then begin
    cdis = " C_DIS"
  end

!---------------------------------------------------------------------------

  if CLD.OnOff=71 and drilling=1 and pos("BodySection", str1)>0 then !only for
hole machining
    drilling = 10

  if CLD.OnOff=72 and drilling=10 and pos("BodySection", str1)>0 then !only for
hole machining
    drilling = 0
```

```
end
```

Access from .NET

Handler:

```
public override void OnStructure(ICLDStructureCommand cmd, CLDArray cld)
```

Inheritance: `ICLDStructureCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOpen`	`bool`	Contains "true" if the command is the start of a new group. Otherwise it contains "false".
`cmd.IsClose`	`bool`	Contains "true" if the command is the end of a group. Otherwise it contains "false".
`cmd.NodeType`	`string`	Textual attribute that defines the type of a group. There are few standard group types like "Approach", "String", "Level" etc. But it can contain any arbitrary value also.
`cmd.Comment`	`string`	Some arbitrary textual commentary or name of the group.

Example:

```
public override void OnStructure(ICLDStructureCommand cmd, CLDArray cld)
{
    if (cmd.IsOpen)
```

```
        nc.OutWithN("(>>> " + cmd.NodeType + ": " + cmd.Comment + ")");    //
group start
    else if (cmd.IsClose)
        nc.OutWithN("(<<< " + cmd.NodeType + ")");                         //
group end
}
```

See also

- CLData access model

SYNCAXES - Axes movement synchronization

The purpose of this command is to enable or disable the synchronization of simultaneous motion of two machine axes (for example left and right spindle rotation axes on a double spindles lathe machine).

Command: SYNCAXES ON(71) | OFF(72), FirstAxisID(Axis1), SecondAxisID(Axis2)

Access from sppx

Handler: `program SyncAxes`

Parameter	CLD array	Description
IsOn	CLD[1]	New state of the synchronization: 71 - Axes synchronization is active. 72 - Axes motion synchronization is disabled.

Parameters available through the cmd operator

Parameter	Type	Description
Action	Integer	Toggle control that modifies the synchronization state (enabled/disabled): 1 - Set synchronization state to enabled. 0 - Set synchronization state to disabled.
Axis1	String	String identifier for the first of the two axes to be synchronized.
Axis2	String	String identifier for the second of the two axes to be synchronized.

Here is a simple example of program handler for this command:

```
program SyncAxes
  Outblock
  a1: String;
  a1 = cmd.Str["Axis1"];
```

```
  a2: String;
  a2 = cmd.Str["Axis2"];
  if (a1="AxisCPos") or (a2="AxisCPos") or (a1="AxisC1Pos") or (a2="AxisC1Pos")
then begin
    if CLD.IsOn=71 then begin
      GSyncSpeeds = 288; GSyncSpeeds@ = Maxreal;
      Formblock;
      Output Outstr$ + " (Sync spindles rotation)";
      Phase = C2-C; Phase@=MaxReal;
      GSyncPhase = 289; GSyncPhase@=0;
      FormBlock;
      output OutStr$ + " (Sync phase)";
    end else begin
      GsyncSpeeds = 290; GsyncSpeeds@ = Maxreal;
      Formblock;
      Output Outstr$ + " (Cancel spindles sync)";
    end;
  end;
end
```

Access from .NET

Handler:

```
public override void OnSyncAxes(ICLDSyncAxesCommand cmd, CLDArray cld)
```

Inheritance: `ICLDSyncAxesCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Contains "true" if the command enables a mode of synchronous movement, otherwise it is "false".
`cmd.IsOff`	`bool`	Contains "true" if the command disables a mode of synchronous

Member	Type	Description
		movement, otherwise it is "false".
`cmd.FirstAxisID`	`string`	Textual axis identifier for the first of the machine axes between which synchronization is performed.
`cmd.SecondAxisID`	`string`	Textual axis identifier for the second of the machine axes between which synchronization is performed.

Example - `OnSyncAxes` (from the Goodway SW postprocessor):

```
public override void OnSyncAxes(ICLDSyncAxesCommand cmd, CLDArray cld)
{
    if (SameText(CurrentOperation.TypeName, "TSTTakeoverMTM")) return;

    base.OnSyncAxes(cmd, cld);
    if (cmd.FirstAxisID.Contains("AxisC1") || cmd.FirstAxisID.Contains("AxisC2")){
        SyncSpindles(cmd.IsOn,false);
    }else
    if (cmd.FirstAxisID.Contains("AxisZ1") || cmd.FirstAxisID.Contains("AxisZ2")){
        SyncZ(cmd.IsOn, false);
    }else{
        Log.Error("Axes can not be syncronized:
("+cmd.FirstAxisID+","+cmd.SecondAxisID+")");
    }
}
```

See also

- CLData access model

TAKEOVER - Workpiece takeover

Command caption

How the command appears in the CLData command list:

```
TAKEOVER WorkpieceConnectorID
```

Use TAKEOVER command to take workpiece over from the active connector into the connector specified in the command parameter. The active workpiece connector is specified by the SELWORKPIECE command. The workpiece connector is a machine unit that is used to fix the workpiece. Use this command for multi-spindle lathe or lathe-milling machines to perform workpiece takeover from one spindle into another.

Access from sppx

Handler: `program Takeover`

The TAKEOVER command's single parameter is the string identifier of the workpiece connector. This identifier is defined by the CAM system machine scheme. Parameter value is stored into the CLDATA$ predefined variable.

Example:

```
program Takeover
  ! CLDATA$ - target workpiece connector identifier
  Output "( TAKEOVER TO " + CLDATA$ + " )"
end
```

Access from .NET

Handler:

```
public override void OnTakeover(ICLDTakeoverCommand cmd, CLDArray cld)
```

Inheritance: `ICLDTakeoverCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.TargetConnectorID`	`string`	Textual identifier of the workpiece connector. This identifier is defined by the machine schema.

Example:

```
public override void OnTakeover(ICLDTakeoverCommand cmd, CLDArray cld)
{
    nc.OutWithN("( TAKEOVER TO " + cmd.TargetConnectorID + " )");   // hand the
workpiece to the target workpiece connector
}
```

See also

- CLData access model

TLCONTACT - Tool contact

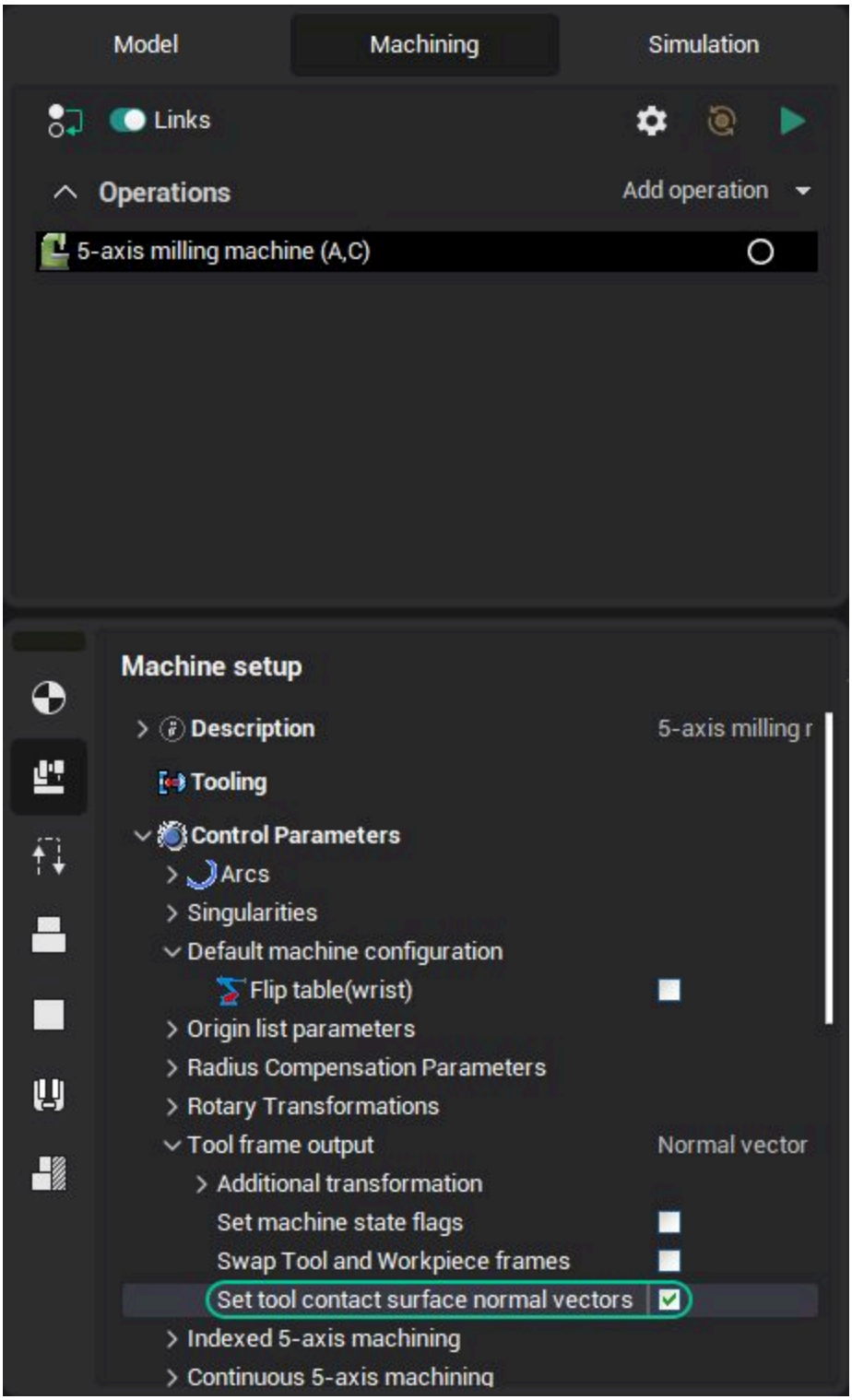
The TLContact command provides information about the normal to the machining surface at the point of contact of the tool with this surface. This is used to calculate 5D correction. In fact, this information determines in which direction (along which vector) the tool needs to be shifted in space by the amount of correction (the difference between the radius of the theoretical and actual tool).

Access from sppx

Handler: `program TLContact`

Parameter	CLD array	CLD name	Description
OnOff	CLD[1]	CLD.OnOff	Determines whether the tool contacts the machined surface in subsequent movements. ON(1) - there is contact information, OFF(0) - there is no contact information.
NX	CLD[2]	CLD.NX	The X component of the surface normal vector
NY	CLD[3]	CLD.NY	The Y component of the surface normal vector
NZ	CLD[4]	CLD.NZ	The Z component of the surface normal vector

This command will be generated in CLData only if the corresponding option "Set tool contact surface normal vectors" is enabled in the settings of the machine schema.



Example:

```
! deltaR - difference between the tool radius set in CAM and the real tool radius on
the machine
! (assign it once, where the real tool radius becomes known)

program TLContact
  ! cld[1] - contact flag ON(1)|OFF(0); cld[2..4] - contact surface normal NX,
NY, NZ
```

```
  if cld[1] = 1 then begin
    NcX = cld[2]  NcY = cld[3]  NcZ = cld[4]             ! remember the contact
normal in globals
    HasContact = 1
    Output "( CONTACT N = " + Str(NcX) + " " + Str(NcY) + " " + Str(NcZ) + " )"
  end else
    HasContact = 0
end

program AbsMov
  ! shift the tool tip along the remembered contact normal by deltaR
  if HasContact = 1 then begin
    X = cld[1] + NcX * deltaR
    Y = cld[2] + NcY * deltaR
    Z = cld[3] + NcZ * deltaR
  end else begin
    X = cld[1]  Y = cld[2]  Z = cld[3]
  end
  OutBlock
end
```

Access from .NET

Handler:

```
public override void OnTLContact(ICLDTLContactCommand cmd, CLDArray cld)
```

Inheritance: `ICLDTLContactCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.IsOn`	`bool`	Determines whether the tool contacts the machined surface in subsequent movements. "True" - there is contact information, "False" - there is no contact information.

Member	Type	Description
`cmd.IsOff`	`bool`	Determines whether the tool contacts the machined surface in subsequent movements. "True" - there is no contact information, "False" - there is contact information.
`cmd.ContactNormal`	`TInp3DPoint`	Machining surface contact normal vector (X, Y, Z).

Example:

```
// deltaR - difference between the tool radius set in CAM and the real tool radius
on the machine
TInp3DPoint contactNormal;
bool hasContact;

public override void OnTLContact(ICLDTLContactCommand cmd, CLDArray cld)
{
    if (cmd.IsOn) {
        contactNormal = cmd.ContactNormal;      // remember the surface
contact normal
        hasContact = true;
        nc.OutWithN($"( CONTACT N = {contactNormal.X} {contactNormal.Y}
{contactNormal.Z} )");
    } else {
        hasContact = false;
    }
}

public override void OnGoto(ICLDGotoCommand cmd, CLDArray cld)
{
    // shift the tool tip along the remembered contact normal by deltaR
(radius correction)
    if (hasContact) {
        nc.X.Show(cmd.EP.X + contactNormal.X * deltaR);
        nc.Y.Show(cmd.EP.Y + contactNormal.Y * deltaR);
        nc.Z.Show(cmd.EP.Z + contactNormal.Z * deltaR);
    } else {
        nc.X.Show(cmd.EP.X);
        nc.Y.Show(cmd.EP.Y);
        nc.Z.Show(cmd.EP.Z);
    }
```

```
    nc.Block.Out();
}
```

See also

- CLData access model

WAIT - Waiting for synchronization point

Command caption

How the command appears in the CLData command list:

```
WAIT ID(<SynchPointID>), Index(<SynchPointIndex>)
```

The WAIT command used to synchronize multiple concurrent threads of control (several NC-programs, each of which can control a separate working element of the machine). For this command should add frames (labels) to NC-program, implementing the suspension of the program until the achievement of a similar label in another (concurrent) program.

Access from sppx

Handler: `program Wait`

Parameters available through the cmd operator

TCLDSyncPoint: ComplexType	Waiting for synchronization point command
SyncPointID: String	cmd.Str["SyncPointID"] - unique identifier of synchronization point, defined by the user.
Index: Integer	cmd.Int["Index"] - ordinal number of synchronization point automatically generated by the system.

When a project uses a «Turn take over» operation that must be synchronized with a «Lathe part-off» operation, the `SyncPointID` parameter in the WAIT commands takes strictly defined values:

- Takeover1: the wait label with this identifier is output immediately after the PPFUN TECHINFO (58) commands of both operations. Both channels wait for mutual synchronization.
- Takeover2: on reaching the wait label with this identifier, the «Turn take over» operation moves to the main spindle and grips the part in it. The «Lathe part-off» operation stays in the waiting state until the wait label with identifier Takeover3 is reached.
- Takeover3: on reaching the wait label with this identifier, the «Lathe part-off» operation cuts the finished part off from the main spindle. The «Turn take over» operation, holding the part being cut off, waits for the cutoff to finish, that is, for the wait label with identifier Takeover4.
- Takeover4: on reaching the wait label with this identifier, the «Turn take over» operation begins retracting the counter-spindle with the gripped cut-off part. The «Lathe part-off» operation waits until

the counter-spindle has been retracted by the Return feed distance, which is marked by the wait label with identifier Takeover5.

- Takeover5: on reaching this wait label, the «Lathe part-off» operation retracts the tool (if required), and the «Turn take over» operation returns the counter-spindle to its initial position.

SyncPointID	«Lathe part-off» operation	«Turn take over» operation
Takeover1	Waiting for mutual synchronization	Waiting for mutual synchronization
Takeover2	Stays in the waiting state until the Takeover3 label is reached.	Moves to the main spindle and grips the part in it.
Takeover3	Cuts the finished part off from the main spindle.	Holding the part being cut off, waits for the cutoff to finish (the Takeover4 label).
Takeover4	Waits until the counter-spindle has been retracted by the Return feed distance (the Takeover5 label).	Begins retracting the counter-spindle with the gripped cut-off part.
Takeover5	Retracts the tool (if required).	Returns the counter-spindle to its initial position.

Example (from the PUMA MX series postprocessor):

```
program Wait
    Call OutBl
    if NextCode= CodeOfCmd("PPFUN") then begin
      output "M01";
      NeedM1 =0;
    end
    i : integer;
    i = Cmd.Int["Index"];
    call AddWait(i);
end
```

Access from .NET

Handler:

```
public override void OnSyncWait(ICLDSyncWaitCommand cmd, CLDArray cld)
```

Inheritance: `ICLDSyncWaitCommand : ICLDCommand`.

Read the parameters from the strongly-typed `cmd` object (recommended); the raw `cld[i]` array (same indices as in sppx) and the named `cmd["..."]` helpers are available too. The table lists the command's members, including those inherited from its base interfaces; the common `ICLDCommand` members (navigation, `CLD`, `CLDFile`, `TechOperation`, ...) are described in the CLData access model.

Member	Type	Description
`cmd.PointIndex`	`int`	Ordinal number of the synchronization point automatically generated by the system.
`cmd.PointID`	`string`	Unique textual identifier of the synchronization point, defined by the user.

Example - `OnSyncWait` (from the HanwhaSwissType postprocessor):

```
public override void OnSyncWait(ICLDSyncWaitCommand cmd, CLDArray cld)
{
    int PntN = 0;
    if (NCFiles.OutputDisabled)
        return;
    if (cmd.PointID.Contains("SubAxial2"))
        return; //Middle label in a wait main spindle operation.
    if (SameText(cmd.PointID, "Takeover1")) return;

    if (SameText(cmd.PointID, "Takeover2")) {
        PntN = 2;
        SynchPoints.OutPrev(cmd.PointIndex,1);
        if (nc==nc1)
        {
            SyncSpindles(true);
        }
    } else if (SameText(cmd.PointID, "Takeover3")) {
        PntN = 3;
        if (activeChannel>0)
          outClamp(false,true);
        SynchPoints.OutPrev(cmd.PointIndex,1);
```

```
        nc.WriteLine(GetSyncZCode(true));
    } else if (SameText(cmd.PointID, "Takeover4")) {
        PntN = 4;
        SynchPoints.OutPrev(cmd.PointIndex,2);
        nc.WriteLine(GetSyncZCode(false));
        if (nc==nc1) {
            SyncSpindles(false);
        } else {
            IncrementalMove=true;
        }
        SynchPoints.OutPrev(cmd.PointIndex,1);
        if (nc==nc1) {
            nc.WriteLine("S0");
        } else {
            nc.WriteLine("S40");
        }
    } else if (SameText(cmd.PointID, "Takeover5")) {
        PntN = 5;
        IncrementalMove=false;
    }

    SynchPoints.Add(cmd.PointIndex);

    if (PntN==4)
        OutDetectBrokenTool();

    if ((nc==nc2) && (PntN==2)) {
        outAirBlower(0);
        outClamp(false, false);
    }
    if ((nc==nc2) && (PntN==5)) {
        nc.WriteLine("M95");
        nc.WriteLine("/M91");
        nc.WriteLine("M96");
    }

}
```

See also

- CLData access model

Contents

The .NET postprocessors subsystem builds postprocessors in .NET (C#). This page lists the contents of the section; the topics below cover the system and its tools, the user interface, and the shared CLData reference read by both postprocessing subsystems.

- External references and examples
- .NET postprocessors
- Introduction
- Organization of the postprocessing system
  - Running the postprocessing
  - Postprocessors' development tools
  - Postprocessor's file set
- User interface
  - CLData Viewer
  - Postprocessor settings window
  - Registers window
- How to write postprocessors
  - How to prepare the computer to start writing the postprocessors
- AI-assisted postprocessor development
  - How the system works
  - Set up the tools
  - Organize your workspace
  - Develop with CAM Agent
  - Develop in Visual Studio Code
  - What the extensions give you
  - SPPX workflow
  - .NET workflow
  - Review and verify
  - Troubleshooting
  - Advanced setup
- CLData reference
  - CLData access functions and operators
    - Named CLData parameters
    - CLD predefined array
    - Current command (Cmd operator)
    - Browsing CLData commands
    - CLData files (CLDFile operator)
    - Project information (Project operator)
    - GMA array
  - Project information in CLData

- Technology commands description
  - AXESBRAKE - Machine axes brakes control
  - CIRCLE - Circle movement
  - CLAMP - Fixtures clamp
  - COMMENT - Commentaries
  - COOLNT - Cooling
  - CUTCOM - Tool compensation
  - CYCLE - Hole machining cycles
    - CYCLE BORE5(209) - G85 drilling cycle
    - CYCLE BORE6(210) - G86 drilling cycle
    - CYCLE BORE7(211) - G87 drilling cycle
    - CYCLE BORE8(212) - G88 drilling cycle
    - CYCLE BORE9(213) - G89 drilling cycle
    - CYCLE BRKCHP(288) - Deep drilling with chip breaking
    - CYCLE DEEP(153) - Deep drilling with chip removing
    - CYCLE DRILL(163) - G81 drilling cycle
    - CYCLE FACE(81) - G82 drilling cycle
    - CYCLE TAP(168) - G84 drilling cycle
  - DELAY - Delay
  - EDMMOVE - EDM movement
  - EFFECTOR - End effector activation
  - EXTCYCLE - Extended cycle
    - EXTCYCLE BRKCHP - Lathe drilling with chip breaking cycle
    - EXTCYCLE DEEP - Lathe drilling with chip removing cycle
    - EXTCYCLE DRILL - Lathe drilling cycle (G81)
    - EXTCYCLE FACE - Lathe drilling cycle (G82)
    - EXTCYCLE LATHEFINISH - Lathe finish cycle (G70, G73)
    - EXTCYCLE LATHEGROOVE - Lathe grooving cycle (G74, G75)
    - EXTCYCLE LATHEROUGH - Lathe roughing cycle (G71, G72)
    - EXTCYCLE LATHETHREAD - Lathe threading cycle (G76)
    - EXTCYCLE LATHETHREADG92(404) - Lathe threading single pass cycle (G92)
    - EXTCYCLE TAP - Lathe tapping cycle (G84)
    - EXTCYCLE W5DBore5 - Drilling cycle (G85)
    - EXTCYCLE W5DBore6 - Drilling cycle (G86)
    - EXTCYCLE W5DBore7 - Drilling cycle (G87)
    - EXTCYCLE W5DBore8 - Drilling cycle (G88)
    - EXTCYCLE W5DBore9 - Drilling cycle (G89)
    - EXTCYCLE W5DChipBreaking - Drilling with chip breaking cycle (G73)
    - EXTCYCLE W5DChipRemoving - Drilling with chip removing cycle (G83)
    - EXTCYCLE W5DDrill - Drilling cycle (G81)

        - EXTCYCLE W5DFace - Drilling cycle (G82)
        - EXTCYCLE W5DHolePocketing - Hole pocketing cycle
        - EXTCYCLE W5DTap - Tapping cycle (G84)
        - EXTCYCLE W5DThreadMill - Thread milling cycle
        - EXTCYCLE WProbing - Probing cycle
            - Additional parameters
            - Boss probing parameters
            - Double wall external corner probing parameters
            - Double wall internal corner probing parameters
            - External rectangle probing parameters
            - Groove probing parameters
            - Groove probing protected parameters
            - Hole probing parameters
            - Hole protected probing parameters
            - Internal rectangle probing parameters
            - Internal rectangle probing protected parameters
            - Surface probing parameters
            - Triple wall external corner probing parameters
            - Triple wall internal corner probing parameters
            - Web probing (three points) parameters
            - Web probing parameters
    - FEDRAT - Feedrate
    - Filter - Output filter (global handler)
    - FINI - Ending record
    - FROM - Original point
    - GOHOME - Return to original position
    - GOTO - Linear transition
    - HEAD - Spindle head number
    - INSERT - Insertion
    - INTERPOLATION - Interpolation mode
    - LOADTL - Tool loading
    - MULTIARC - Multi-axis circle movement
    - MULTIGOTO - Multi-coordinate movement
    - OPSKIP - Optional skipping
    - OPSTOP - Auxiliary stop
    - ORIGIN - Original coordinates
    - PALETA - Palette changing
    - PARTNO - Part number
    - PHYSICGOTO - Physical machine axes movement
    - PLANE - Working plane

- POWER - Power adjustment
- PPFUN - Postprocessor function
  - PPFUN CALLSUB(52) - NC-subprogram call command
  - PPFUN ENDSUB(51) - NC-subprogram end command
  - PPFUN ENDTECHINFO(59) - End of operation's technological information
  - PPFUN PPFUN(500) - Tool durability control command
  - PPFUN STARTSUB(50) - NC-subprogram start command
  - PPFUN TECHINFO(58) - Technology parameters command
  - PPFUN WEDMConditions(56) - EDM machining conditions specification command
  - PPFUN WEDMPassItem(57) - Wire EDM pass properties
- PPRINT - Postprocessor printing
- RAPID - Rapid feedrate
- ROTABL - Table rotation
- SAFPOS - Tool change point
- SELWORKPIECE - Active workpiece connector selection
- SINGLETHREAD - Single-pass thread
- SPINDL - Spindle
- STOP - Stop
- STRUCTURE - Toolpath structure
- SYNCAXES - Axes movement synchronization
- TAKEOVER - Workpiece takeover
- TLCONTACT - Tool contact
- WAIT - Waiting for synchronization point

External references and examples (.NET postprocessors)

External links, the generated SDK API reference, and example postprocessors for the .NET postprocessing subsystem.

Product documentation

Reference	URL
ENCY portal	https://encycam.com/
.NET postprocessing guide	Open the local guide
ENCY Academy — training courses	https://learn.encycam.com/

AI-assisted development

Download locations for the tools described in AI-assisted postprocessor development.

Reference	URL
Postprocessor tools for VS Code — releases	https://github.com/EncySoftware/postprocessor-tools/releases
ENCY documentation center	https://docs.encycam.com/portal/

API reference

The .NET postprocessing SDK API reference is generated automatically from the source code.

- .NET postprocessing SDK API reference

External links

Reference	URL
.NET platform	https://dotnet.microsoft.com/
.NET SDK download	https://dotnet.microsoft.com/download
C# programming language documentation	https://docs.microsoft.com/en-us/dotnet/csharp/

Reference	URL
Visual Studio Code	https://code.visualstudio.com/
Visual Studio Code — Get started	https://code.visualstudio.com/docs/getstarted/userinterface
C# extension for VS Code	https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp
Nuget package repository	https://www.nuget.org/

Example postprocessors

A set of example postprocessors is shipped together with the CAM system, in the Dotnet postprocessors folder. You can open any of them in the CLData Viewer and Visual Studio Code, study the source code, build it and use it as a starting point for your own postprocessor.

.NET postprocessors

This is the open postprocessing system based on the .NET platform. It uses C# as a programming language and Visual Studio Code as the development environment and code debugger.

The CAM system has two independent postprocessing subsystems; both are actual and can be used by your choice. This section documents only the .NET based system. The other subsystem (the Postprocessors generator) is documented under Postprocessors generator.

Why this system

- Based on widespread, free and open-source top-class tools: .NET, C# and Visual Studio Code. All modern programming technologies are available to postprocessor developers.
  - The power and at the same time the simplicity of the C# programming language: object-oriented style, strong and safe typing, automatic garbage collection, and features like generics, lambda expressions, LINQ and tuples.
  - Syntax highlighting and autocomplete with IntelliSense.
  - A first-class debugger with different kinds of breakpoints, call stacks, an interactive console, a local variables view and the watch list.
  - A high-end code editor with a huge number of useful extensions for all occasions.
  - An unlimited amount of documentation available on the network.
  - Postprocessor documentation available directly in the code as comments and pop-up tips.
- The familiar register-based concept of the Postprocessors generator, re-implemented from scratch on a new level with a modern object-oriented paradigm and taking into account decades of experience in postprocessor development. This provides advantages such as:
  - Easy writing of postprocessors that generate many files at the same time.
  - The ability to append to an arbitrary location in a file, not just to the end.
  - Generating output files in any encoding.
  - Multiple independent sets of registers.
  - Text registers in addition to numeric registers.
  - Developer-defined settings individual for each postprocessor.
  - An integrated library of functions for geometric calculations.
  - The ability to use third-party libraries widely available through Nuget.

Documentation

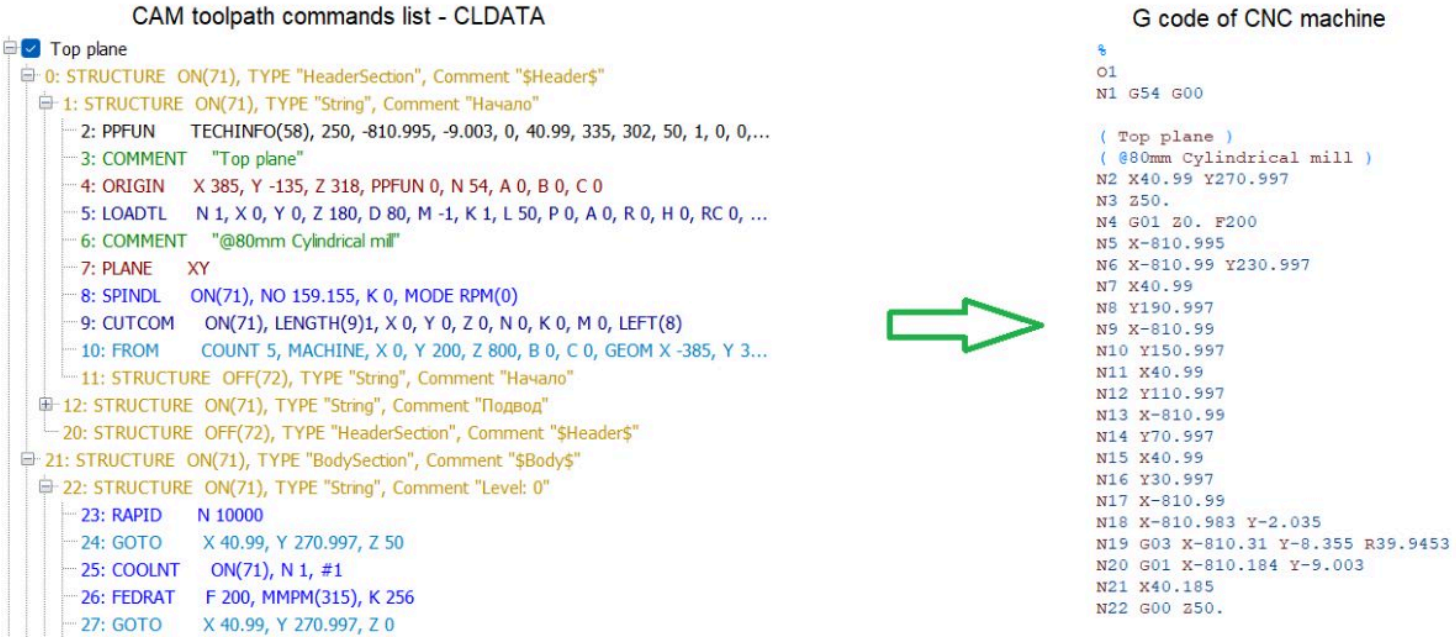
- Introduction — what a postprocessing system is and the main program modules.
- Organization of the postprocessing system
  - Running the postprocessing
  - Postprocessors' development tools
  - Postprocessor's file set
- User interface

    - CLData Viewer
    - Postprocessor settings window
    - Registers window
    - Visual Studio Code user interface
- How to write postprocessors
    - How to prepare the computer to start writing the postprocessors
- CLData documentation — the shared input model read by both postprocessing subsystems.
- External references and examples — the .NET SDK API reference, external links and example postprocessors.

Introduction

What is a postprocessing system in general?

A postprocessing system is a program that converts a toolpath generated by the CAM system from a universal format (CLData) into output files understandable by a specific CNC machine.



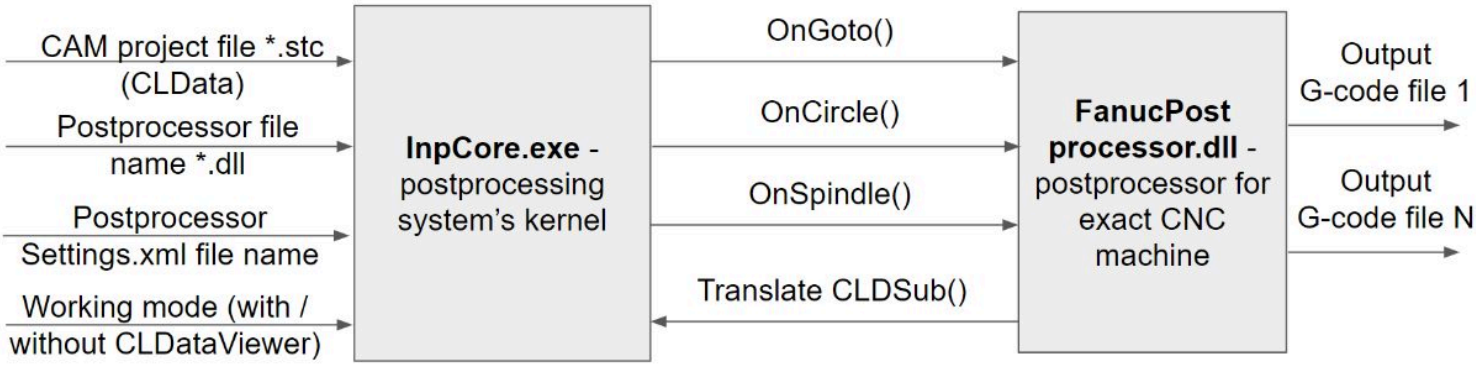
CLData (cutter location data) is a set of commands like tool loading, movements, spindle and coolant switch on/off, etc.

A postprocessing program gets it as an input and converts it to the format which is required by the exact CNC machine, for example a G-code text file. There are a lot of CNC machines and each type of machine can use its own format of files. Therefore the postprocessing system should contain at least two parts.

- The first is the part independent from the exact type of machine. It must be able to read the input CLData and provide it for further convenient processing.
- The second part is the adjustable part. It should provide a flexible way to translate the universal commands into the specific output files.

Main program modules of the system

From the point of view of program modules, it looks like the one shown below.



The first main starting module is an InpCore.exe utility. It's a console application which means that it has no user interface, it has only a console window. Inside this console window you can type some command line parameters like the name of the input CLData file (stc-project), the name of the postprocessor and some additional settings (output file name, program number etc.) and then run the process of translation. This utility reads the input CLData file and starts to call a special kind of subroutines — command handlers — which are individual for each command type. For example for the LoadTool CLData command it will call the subroutine with the OnLoadTool name, for the Spindle command — the OnSpindle subroutine, for the Goto command — the OnGoto subroutine etc. By default these command handlers are empty and do nothing. To be able to override this empty behavior we need to load the second part of the postprocessing system — the postprocessor itself.

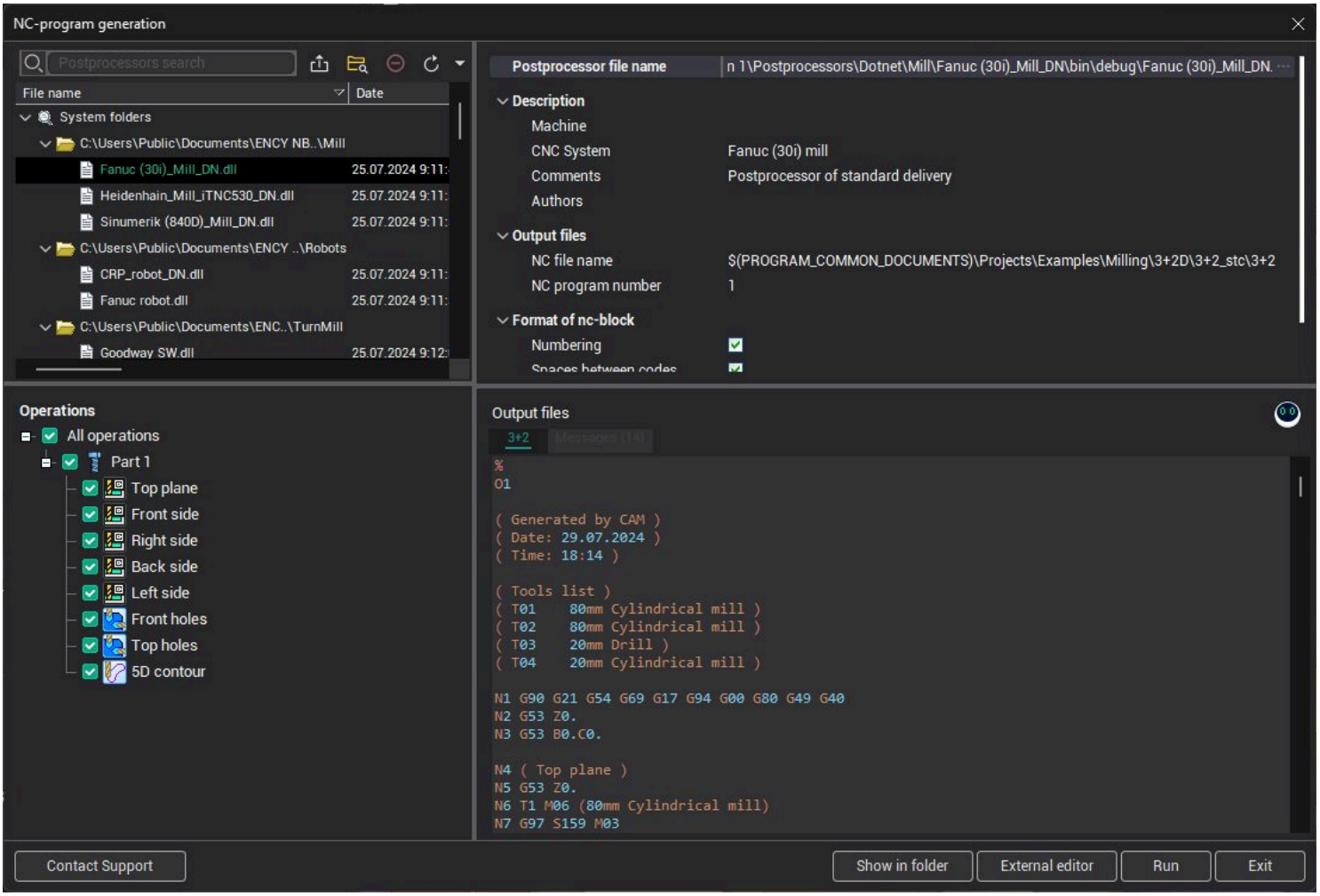
The postprocessor is an additional dynamically loaded program module with the dll extension. The name can vary. For example FanucPostprocessor.dll. It can implement some of the command handlers which will override the empty behavior and will write to the output file specific instructions corresponding to the command for the exact CNC machine. We can easily change the postprocessor to another one, for example HeidenhainPostprocessor.dll, to get the resulting G-code for another machine. You can make your own postprocessor if you want to have your unique output file content.

Running the postprocessing

Let's see how to generate a G-code using the postprocessors.

NC-program generation window

The first, simplest way is to use the window of the CAM system specially designed for this. You can open it using the "Postprocessor" button at the top toolbar of the main window.

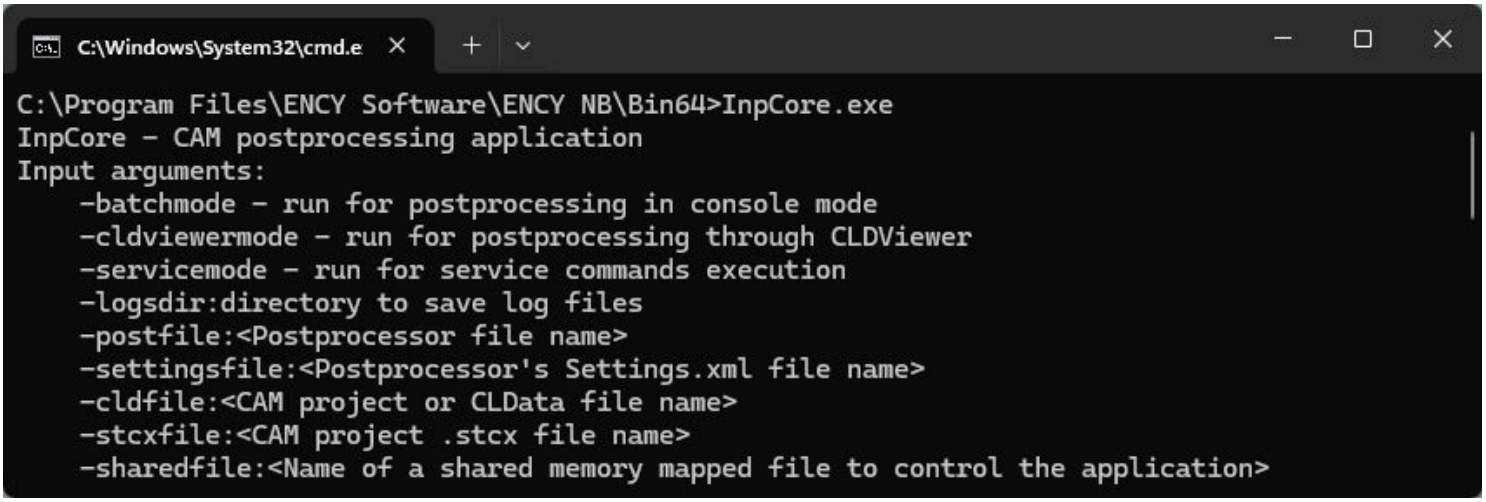


Here you just need to choose the desired postprocessor in the left top list, then define some additional settings in the right top properties list like the output NC file name, for example, and, finally, click the "Run" button. At the bottom part of the window you will see the content of a generated G-code file. To get the file itself you can use the "Show file in Windows explorer" popup menu item of the panel with the G-code.

Running of postprocessors in console mode

The postprocessing system is a standalone program, so you can also run it independently from the CAM system, the same way the CAM system does it behind the scenes. To do it just run the standard Windows command prompt from the Start menu or "Win+R" keyboard shortcut and type "cmd". Here you

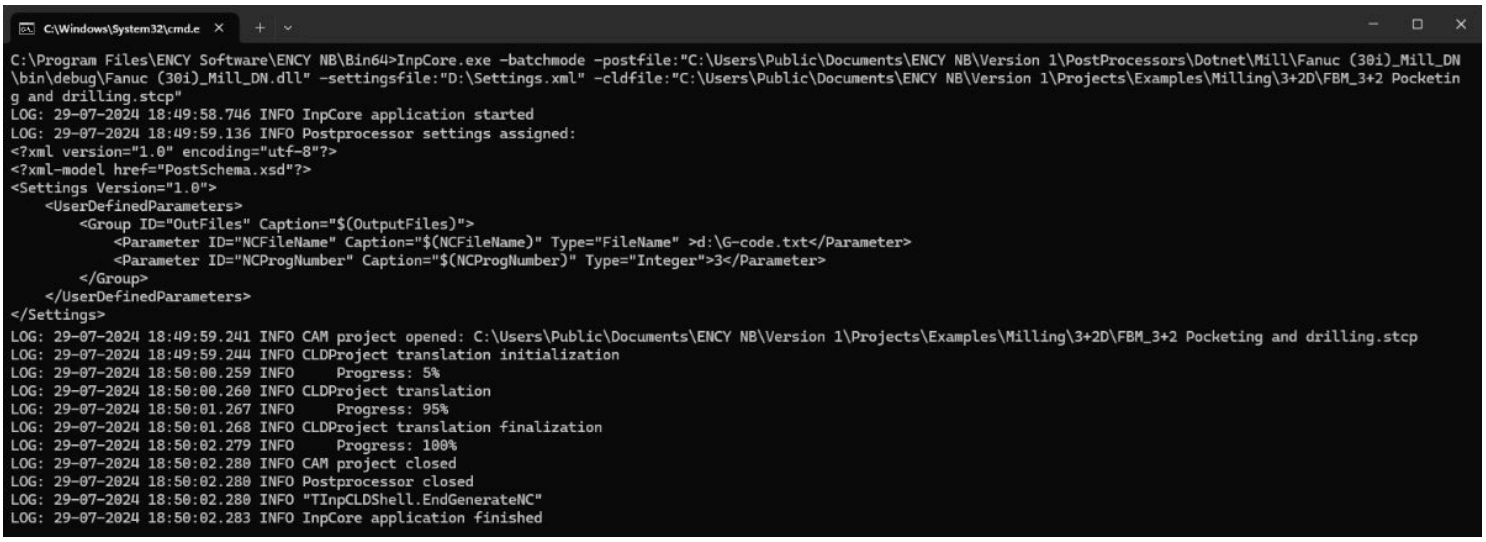
need to type the path to the InpCore.exe utility from the CAM installation folder. It will show you the welcome screen with the list of available parameters.



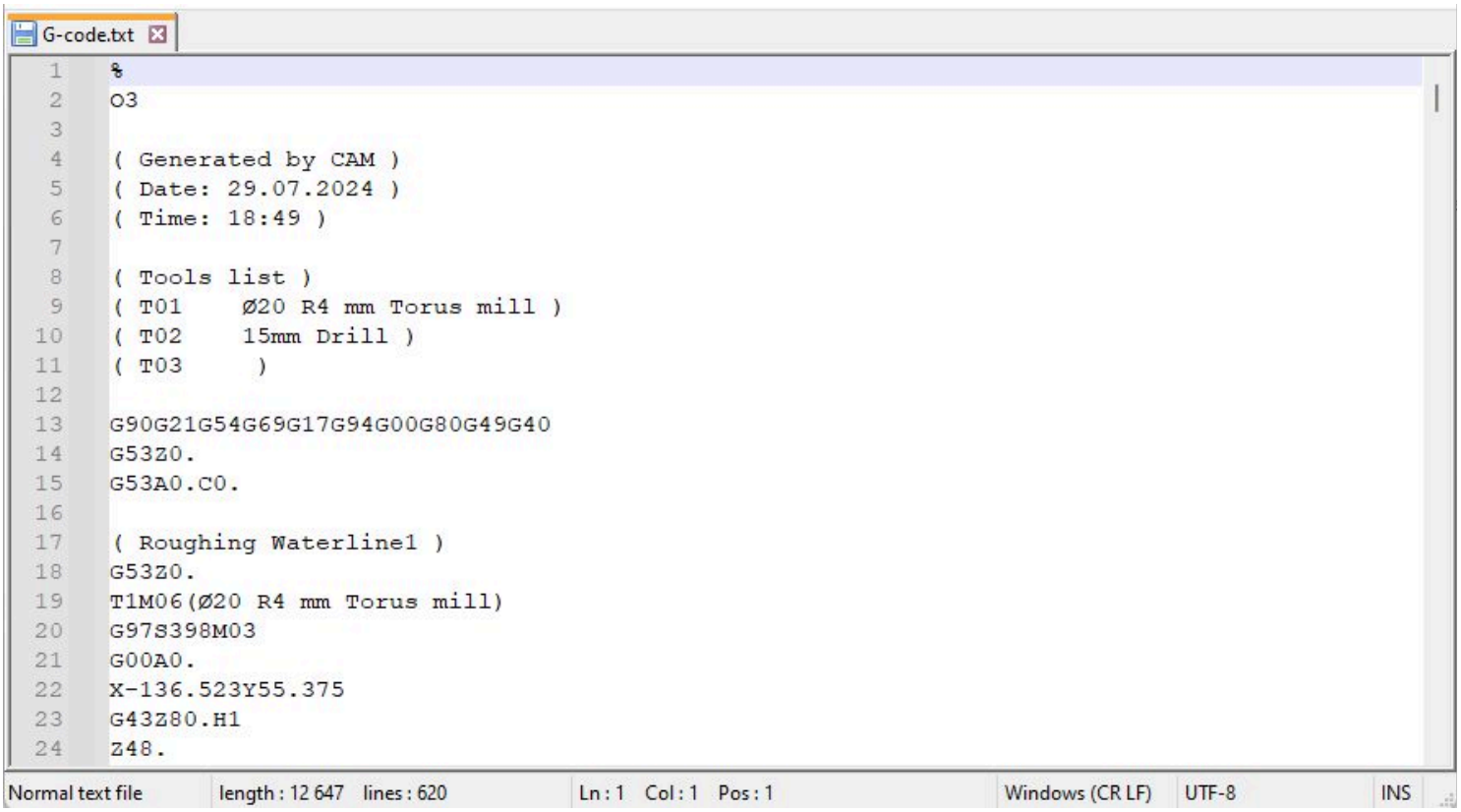
To generate a G-code you should specify the following parameters.

- "-batchmode" — to start standalone postprocessing without postprocessor debugging through the CLData Viewer.
- "-postfile:`<path to the postprocessor dll>`" — full path to the postprocessor's dll file you want to use.
- "-cldfile:`<path to the CAM project stcp>`" — full path to the CAM project stcp file which contains the toolpath you want to postprocess.
- "settingsfile:`<path to the postprocessor's settings.xml file>`" — the full path to the xml file, which should contain settings specific to a particular postprocessor. For example, inside it you can specify the output file name, G-code program number, employee name, etc.

Below is an example of postprocessing in console mode.



And the resulting G-code.



Postprocessors' development tools

The postprocessor is a regular program module with the dll extension, so to be able to make your own postprocessors you should use program development tools. In order to make the development of postprocessors widely accessible, simple and convenient, this system is built on the basis of well-known, free and convenient tools.

.NET platform

The .NET (dotnet) platform is a free, open-source and cross-platform toolkit that allows you to create various types of applications. To start building .NET apps you need to download and install the .NET SDK (Software Development Kit) — a set of utilities which helps you compile binary program modules from the source files, and also contains a rich set of standard libraries to simplify the process of program creation, because it provides a large number of ready-made solutions for performing routine tasks (working with files, strings, lists, etc.). .NET is a very flexible and extensible tool, as it allows you to easily include third-party libraries from the several million available in the global nuget repository. If you want to implement something ordinary (create a standard doc or pdf file, read a database file, create a beautiful html report, etc.), then with a high degree of probability someone else has already done it before you — just use it.

Visual Studio Code

Visual Studio Code, also commonly referred to as VS Code, is a free, lightweight and powerful source-code editor made by Microsoft. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, and code refactoring. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality. Visual Studio Code is one of the most popular developer environment tools around the world.

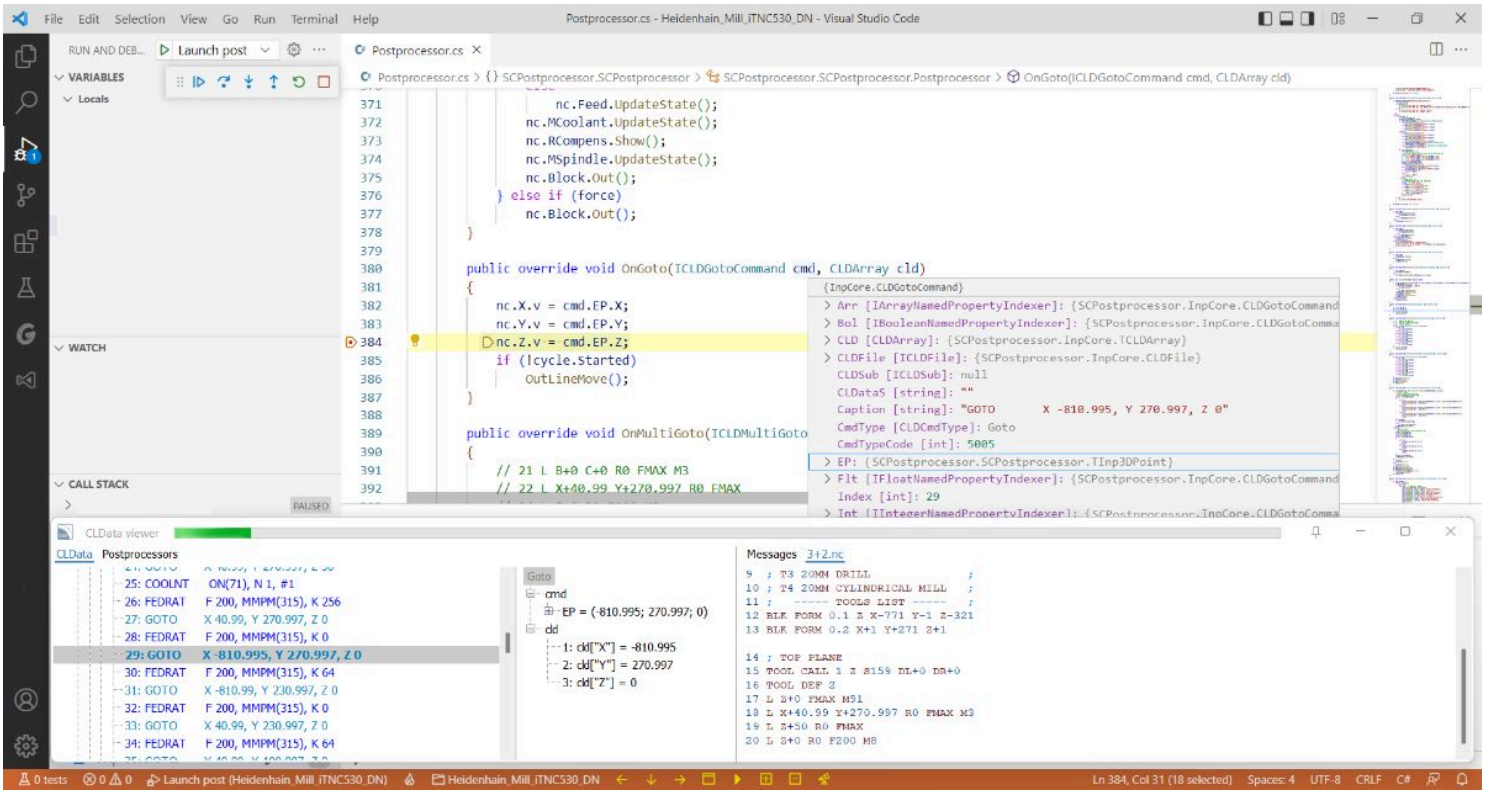
C# programming language

The C# (C-sharp) programming language is a simple, modern, object-oriented, general-purpose programming language. It is in the top 5 languages according to generally recognized ratings (like TIOBE, PYPL). There is a huge amount of training material and answers to frequently asked questions on the network. To switch on support of `C#` in VS Code you need to install the C# extension from the marketplace.

CLData Viewer

CLData Viewer is a window program, specially designed by the CAM developers for the purpose of debugging postprocessors. It allows you to view the toolpath from the CAM project in the form of a list of CLData commands and their parameters. Right during debugging the code of a postprocessor in VS Code, CLData Viewer tracks the output files in real time. It allows you to jump from a command to a line

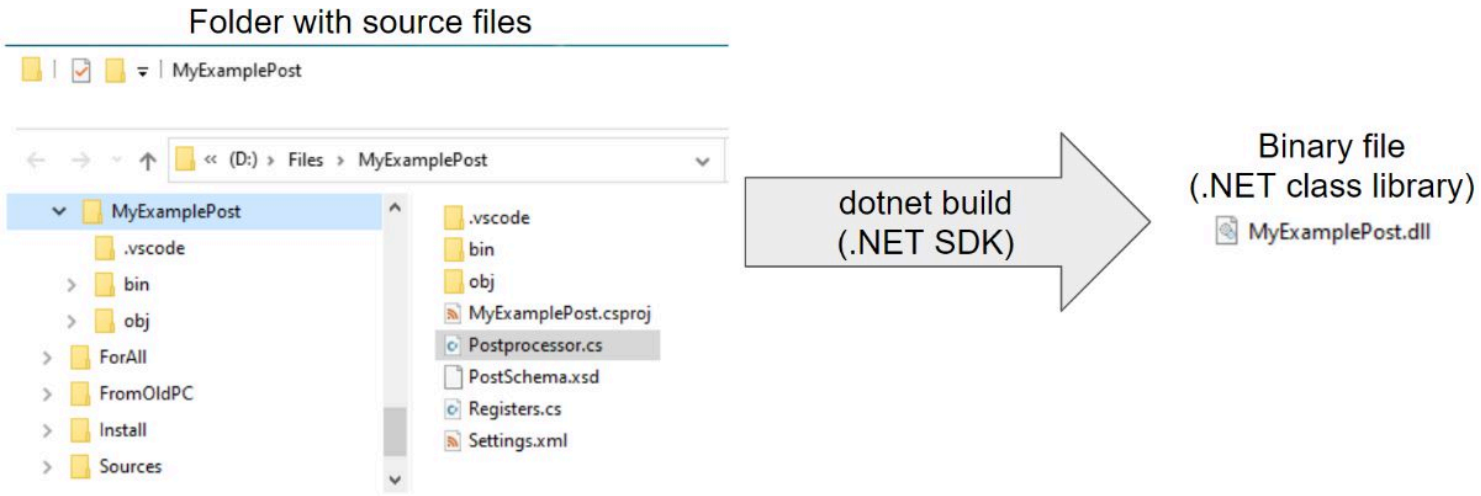
in the resulting output file and back. It supports CLData breakpoints, helps to create new postprocessors and edit their settings.



Postprocessor's file set

Each postprocessor is a compiled program consisting of several files. All files related to one postprocessor are usually located in one folder. The set of files can be divided into two main parts.

- Source code files are text files containing program code that is written by the postprocessor developer using the C# programming language. Source files are only needed on the developer's side of the postprocessor. Usually they are not passed to the client (the one who will use the postprocessor to get the G-code).
- Postprocessor binary file with the *.dll extension, which is produced by the dotnet platform compiler based on the source files. The compiler usually produces many binaries, most of which are auxiliary in nature. There is only one main resulting binary dll file — the file whose name is the same as the postprocessor folder name (and also the *.csproj project file name). This file is self-sufficient for generating G-code, so only it needs to be passed to the client.



Let's take as an example the contents of a folder, which is a set of files of one postprocessor. Here blue color are folders, green color are source text files, red is the main resulting binary file, gray are auxiliary binary and text files.

- MyExamplePost — the main folder of the postprocessor. The name of this folder defines the name of the postprocessor. It should be the same as the name of the .csproj file (if you want to rename the post you should rename both — the folder and .csproj file).
  - MyExamplePost.csproj — C# project file of the postprocessor. It does not contain meaningful postprocessor code. It simply describes that it is a regular C# program. Usually you don't need to change anything in this file.
  - Postprocessor.cs — the main source file of the postprocessor. This is exactly the file that contains meaningful program code that determines the logic of the postprocessor. The developer should write their code here.
  - Registers.cs — an auxiliary source file which contains declarations of the registers (NCWords) that can be used to format and filter text before outputting it to G-code. Do not place meaningful

code here, use it to declare registers only. You can change it manually or using the registers editing window of CLData Viewer.

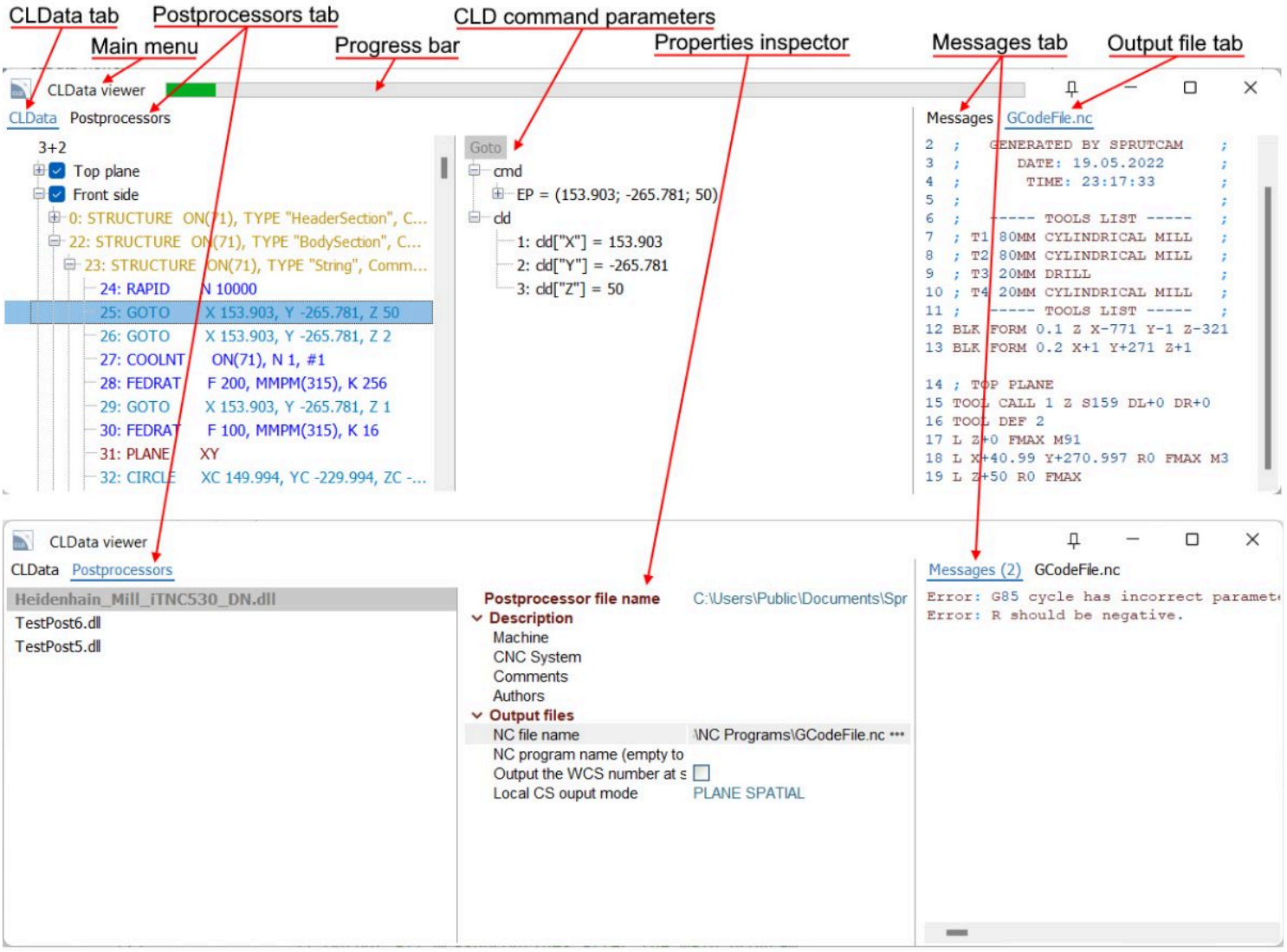
- Settings.xml — this file contains the descriptive properties of the postprocessor (authors, CNC and machine name), as well as a list of additional properties created by the developer, which are shown to the client when using the postprocessor and which they must fill in before starting the generation of G-code (for example, the output file name, program number, username, etc.). You should use the "Edit postprocessor XML settings" window of the CLData Viewer to change this file.
- PostSchema.xsd — auxiliary file that describes the syntax rules for the Settings.xml file. It is needed for error highlighting and auto-completion when manually editing the Settings.xml file. You don't need to change the PostSchema.xsd file.
- .vscode — the folder which contains settings for Visual Studio Code to automate the compilation and debug process of the postprocessor. Usually you don't need to change anything here.
  - launch.json — the file contains the shell command which should be executed to start debugging of the postprocessor when you press the "F5" key.
  - tasks.json — the file contains the shell command which should be executed to run building of the postprocessor when you press "Shift+Ctrl+B".
- bin\Debug — this folder contains a few binaries automatically generated during compilation. If they are deleted, they will be automatically restored at the next compilation.
  - MyExamplePost.dll — it is the main resulting self-sufficient output binary file of the postprocessor. For the client, you must supply only this one file.
  - SCPostprocessor.dll — the standard CAM library which contains the common description of postprocessing types and classes. It is automatically copied from the CAM installation folder during build.
  - STDefLib.dll — the standard CAM library which contains common purpose helper functions which can be used by postprocessors. It is automatically copied from the CAM installation folder during build.
  - VecMatrLib.dll — the standard CAM library which contains helper functions to work with geometry which can be used by postprocessors. It is automatically copied from the CAM installation folder during build.
  - MyExamplePost.deps.json — this file describes which additional libraries the postprocessor depends on.
  - MyExamplePost.pdb — debug symbols file.
  - SCPostprocessor.pdb — debug symbols file.
  - STDefLib.pdb — debug symbols file.
  - VecMatrLib.pdb — debug symbols file.
  - SCPostprocessor.xml — tooltip documentation text file.
  - STDefLib.xml — tooltip documentation text file.
- obj — autogenerated folder.

- a lot of auxiliary autogenerated files. If they are deleted, they will be automatically restored at the next compilation.

CLData Viewer's main window

CLData Viewer

CLData Viewer is a window program, specially designed by the CAM developers for the purpose of debugging postprocessors. It allows you to view the toolpath from the CAM project in the form of a list of CLData commands and their parameters. Right during debugging the code of a postprocessor in VS Code, CLData Viewer tracks the output files in real time. It allows you to jump from a command to a line in the resulting output file and back. It supports CLData breakpoints, helps to create new postprocessors and edit their settings.



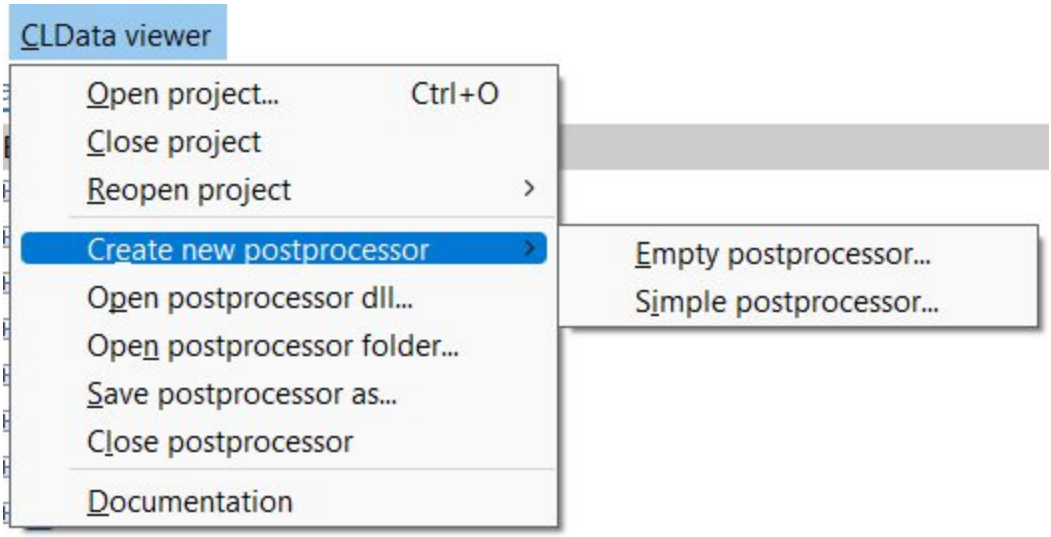
The main window of the CLData Viewer contains the following items.

- Main menu. It contains the main actions you can do in the CLData Viewer like opening the projects and postprocessors.
- CLData tab — the list with the CLData commands of the currently opened CAM system's project. Use the popup menu to operate with projects and breakpoints on CLData commands.

- CLD command parameters panel shows the list of main parameters of the CLData command which is selected on the CLData tab. Using the popup menu of this panel, you can easily get the name of the parameter by which you can access it from the program code of the postprocessor.
- Postprocessors tab — the list of postprocessors opened in CLData Viewer. Here, the postprocessor that was debugged last in VS Code is highlighted in bold. Use the popup menu of this panel to perform actions on the postprocessors.
- Properties inspector shows the properties of the postprocessor which is selected on the postprocessors tab. Here you can define some values which will be the input for the postprocessor to generate the G-code output files. For example, it can be the name of the output file, program number, etc.
- Output file tab — shows the content of the output file (G-code).
- Messages tab — displays a list of messages (errors, warnings, infos) that the postprocessor generates during work.
- Progress bar — shows the progress of the postprocessing (what percentage of the total number of CLData commands has already been handled).

Main menu

The main menu of CLData Viewer looks like the one shown in the picture below.



- Open project... — allows you to open the CAM system project (*.stcp file) with a CLData which will be used to debug postprocessors. A standard file open dialog will be shown.
- Close project — closes the opened CAM system project.
- Reopen project — shows the list of recently loaded CAM system projects to open them quickly.
- Create new postprocessor — helps to create a new postprocessor based on the selected template.
- Open postprocessor dll.. — allows you to open an existing postprocessor in the CLData Viewer by choosing a *.dll file of this postprocessor. A standard file open dialog will be shown.
- Open postprocessor folder.. — allows you to open an existing postprocessor in the CLData Viewer by choosing a folder of this postprocessor. A standard folder open dialog will be shown.

- Save postprocessor as... — here you can save the selected postprocessor with another name.
- Close postprocessor — removes the currently selected postprocessor from the list of postprocessors opened in CLData Viewer.
- Documentation — opens the online documentation of postprocessing.

CLData tab

The CLData tab shows a tree with the CLData commands of the currently opened CAM system's project.

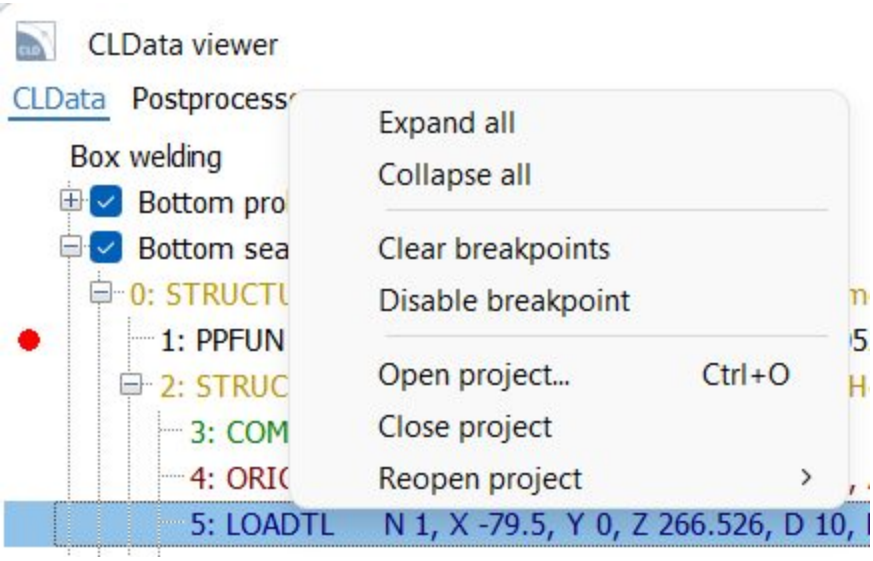
Top level items are so-called "CLData files" which can represent technological operations of the CAM system or CLD-subroutines.

The nested nodes are CLData commands which represent a toolpath of each operation, calculated by the CAM system. The color of the node denotes the command type. The currently postprocessed command when debug is in progress is shown in bold. A detailed list of parameters for the selected command is shown on a separate panel to the right of the CLData list. Using the popup menu of this panel, you can easily get the name of the parameter by which you can access it from the program code of the postprocessor. Also you can copy to clipboard the value of any parameter.

Double click on a CLData command will show you the lines inside the output file which were generated by this command.

There is a concept of so-called "CLData breakpoints". By clicking on a gutter (free space to the left of the CLData commands) next to the desired command you can activate a special kind of breakpoint. It will be shown as a bold red point there. It means that the postprocessing should be paused before handling of this command and you can explore the process of program execution for this command in more detail. This works in conjunction with the "Function breakpoint" feature of Visual Studio Code only. On the breakpoints panel ("Run and debug" tab) of Visual Studio Code you can add a new breakpoint and specify the name of the function you want to stop on. Here you should specify the special function name "StopOnCLData". In this case the postprocessing system will go into the function with such a name each time when it reaches the CLData command that is marked with a CLData breakpoint. Of course the program code of your postprocessor should have (override) the function with this name. By default any postprocessor already has it if you created it using standard templates.

The popup menu of the CLData tab looks like this one.

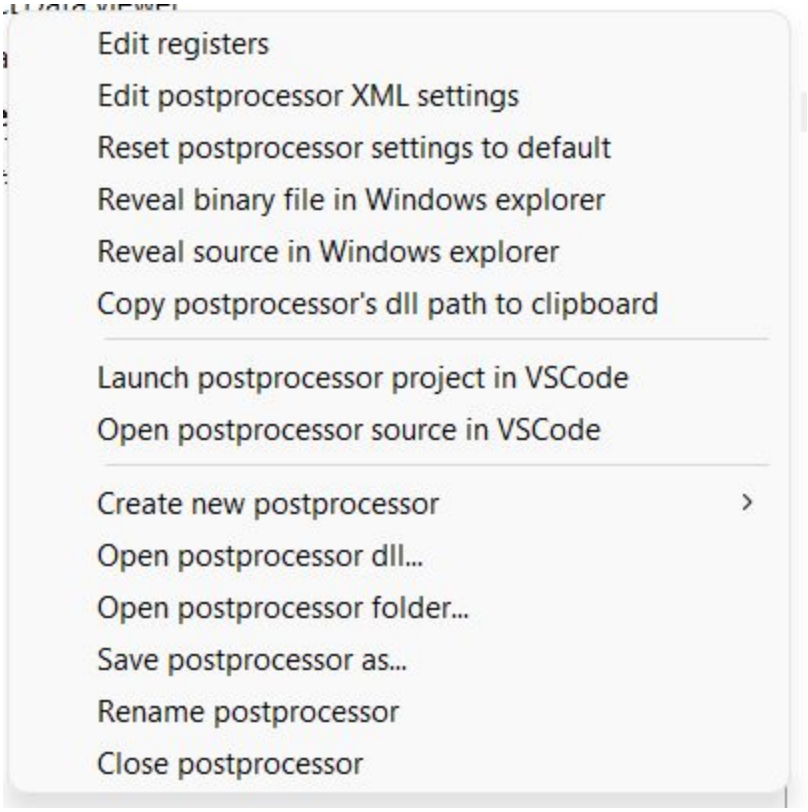


- Expand all — expands all CLData nodes in the tree.
- Collapse all — collapses all CLData nodes in the tree.
- Clear breakpoints — deletes all CLData breakpoints from CLData Viewer.
- Disable breakpoint — temporarily deactivates a CLData breakpoint on the selected command until you enable it again.
- Open project... — allows you to open the CAM system project (*.stcp file) with a CLData which will be used to debug postprocessors. A standard file open dialog will be shown.
- Close project — closes the opened CAM system project.
- Reopen project — shows the list of recently loaded CAM system projects to open them quickly.

Postprocessors tab

The postprocessors tab allows you to view and edit parameters of several postprocessors opened in the CLData Viewer. The left part shows the list of postprocessors, the right part shows an inspector with parameters of the selected postprocessor. The postprocessor that was debugged last in Visual Studio Code is highlighted in bold in the list.

You should use the popup menu to perform actions on the postprocessors.



- Edit registers — opens the window to edit the registers list of the selected postprocessor.
- Edit postprocessor XML settings — opens the window to edit the list of settings of the selected postprocessor.
- Reset postprocessor settings to default — restores the default value for each parameter of the selected postprocessor which is defined in the "settings.xml" file of it. When you type some values in the inspector of properties (for example an output file name) it affects the debug process only and is not saved inside the postprocessor itself. To change the default values use the window to edit the list of settings or edit the file manually.
- Reveal binary file in Windows explorer — opens the Windows explorer in the folder where the resulting binary *.dll file of the postprocessor is placed and selects it.
- Reveal source in Windows explorer — opens the Windows explorer in the folder where the source code Postprocessor.cs file of the postprocessor is placed and selects it.
- Copy postprocessor's dll path to clipboard — copies to clipboard the path to the resulting binary *.dll file of the selected postprocessor.
- Launch postprocessor project in VSCode — starts Visual Studio Code (or activates an existing window) and opens the folder of the selected postprocessor in it. After that you can run and debug it. Or alternatively you can just double click on the postprocessor to do the same.
- Open postprocessor source in VSCode — starts Visual Studio Code (or activates an existing window) and opens the source code file Postprocessor.cs of the selected postprocessor in it as an auxiliary additional file tab exactly. It does not close the project currently opened in VS Code, so it will not allow you to run and debug this postprocessor (because you should open it as a folder to be able to debug), but you can view it. It can be useful when you want to compare the source code of several postprocessors and copy it from one to another.

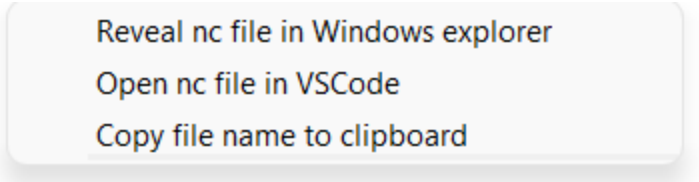
- Create new postprocessor — helps to create a new postprocessor based on the selected template.
- Open postprocessor dll.. — allows you to open an existing postprocessor in the CLData Viewer by choosing a *.dll file of this postprocessor. A standard file open dialog will be shown.
- Open postprocessor folder.. — allows you to open an existing postprocessor in the CLData Viewer by choosing a folder of this postprocessor. A standard folder open dialog will be shown.
- Save postprocessor as... — here you can save the selected postprocessor with another name.
- Rename postprocessor — allows you to easily and safely rename the selected postprocessor.
- Close postprocessor — removes the currently selected postprocessor from the list of postprocessors opened in CLData Viewer.

Output files and messages tabs

These tabs can show you the results of work after the run of a postprocessor in Visual Studio Code.

- Output file tab — shows the content of the output file (G-code). There may be several file tabs if the postprocessor generates many files at the same time. It updates dynamically while the postprocessor is running and debugging. Double click on a line in the file will show you the CLData command which generated this line.
- Messages tab — displays a list of messages (errors, warnings, infos) that the postprocessor generates during work.

The popup menu of this panel contains the following items.



- Reveal nc file in Windows explorer — opens the Windows explorer in the folder where the resulting output file is placed and selects it.
- Open nc file in VSCode — starts Visual Studio Code (or activates an existing window) and opens the resulting output file in it as an auxiliary additional file tab. It does not close the project currently opened in VS Code.
- Copy file name to clipboard — copies the name of the selected file with full path to clipboard.

Postprocessor settings window

What are settings?

Each postprocessor has some set of parameters which we can divide into three main groups.

1. Descriptional properties like Machine and CNC system name, comments and list of authors.
2. Common properties which define the way how to handle some frequently encountered cases. For example, should it break circular arc movements into lines, halves or quarters.
3. User defined parameters — they are a list of additional properties designed by the developer of an exact postprocessor, which are shown to the user of the postprocessor and which they must fill in before starting the generation of G-code. For example, the path to an output file, program name or number, employee name, etc.

The postprocessor stores its settings in a separate settings.xml file which is placed in the folder of source files next to the main postprocessor.cs file. When compiling, this file is embedded inside the main binary dll file of the postprocessor.

Example of Settings.xml file content

Below is an example of the content of a settings.xml file for a real Fanuc robot postprocessor.

▼ Settings.xml — click to expand.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-model href="PostSchema.xsd"?>
<Settings Version="1.0">
  <Description>
    <Comments />
    <CNCSystem />
    <MachineName />
    <Authors />
  </Description>
  <Common>
    <BreakArcs>No</BreakArcs>
    <!-- No, Cuts, Halfs, Quaters -->
    <BreakHelix>false</BreakHelix>
    <!-- false, true -->
    <ArcTolerance>0.01</ArcTolerance>
    <MaxCircleRadius>0</MaxCircleRadius>
    <DecimalSeparator>.</DecimalSeparator>
  </Common>
  <UserDefinedParameters>
    <Group ID="OutFiles" Caption="$(OutputFiles)">
      <Parameter ID="OutputFolder" Caption="$(NCFilesDir)" Type="FolderName" />
```

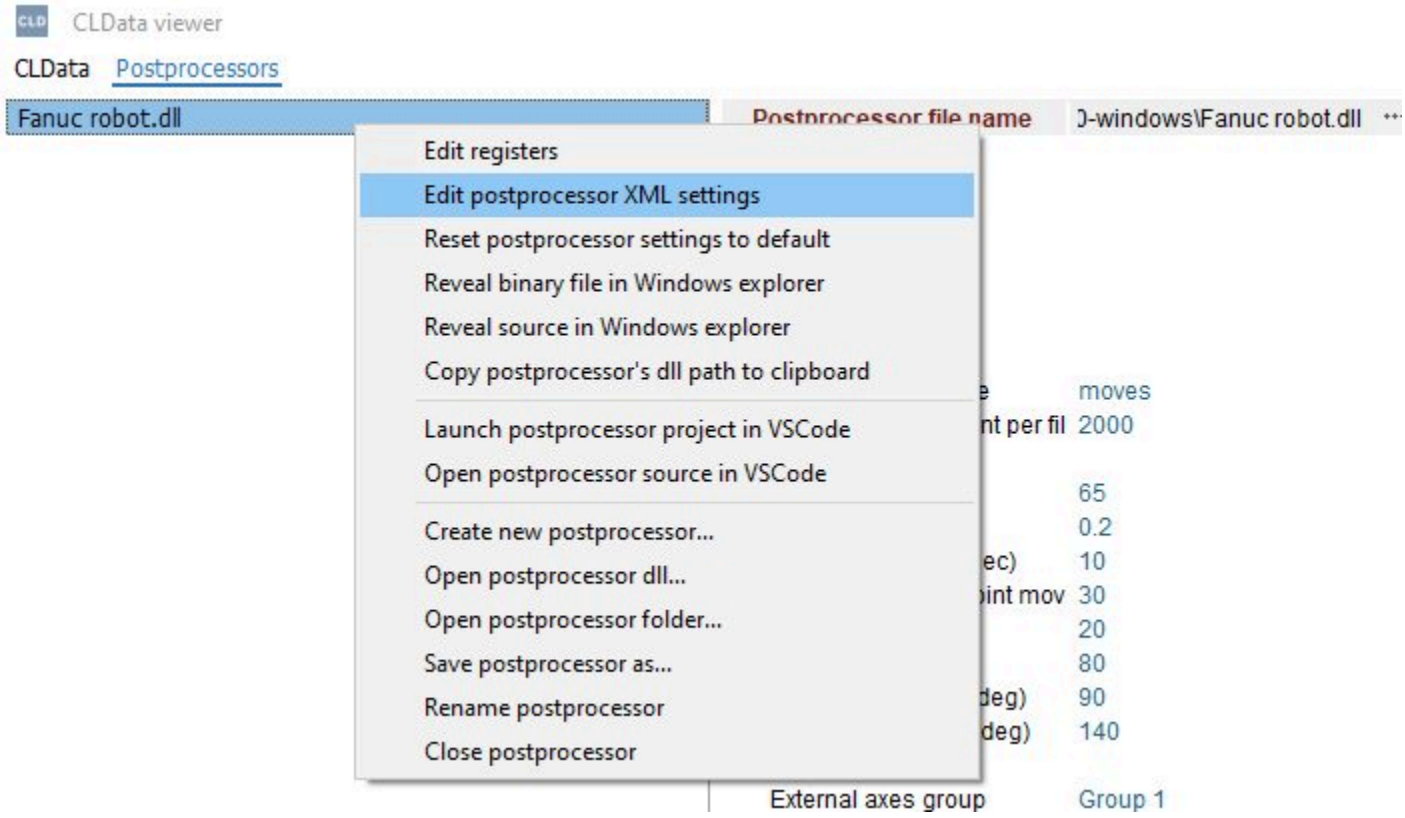
```
      <Parameter ID="LSFileName" Caption="*.ls files short name"
Type="String">moves</Parameter>
      <Parameter ID="MaxMoveCount" Caption="Max movement count per file"
Type="Integer">2000</Parameter>
    </Group>
    <Group ID="Smoothing" Caption="Smoothing">
      <Parameter ID="DefaultACC" Caption="Default ACC value"
Type="Integer">65</Parameter>
      <Parameter ID="ShortLength" Caption="Short length"
Type="Double">0.2</Parameter>
      <Parameter ID="StartVelocity" Caption="Start velocity
(mm/sec)" Type="Double">10</Parameter>
      <Parameter ID="CntValueRapid" Caption="CNT for Rapid or Joint move (%)"
Type="Integer">30</Parameter>
      <Parameter ID="CntValueMin" Caption="Min CNT value (%)"
Type="Integer">20</Parameter>
      <Parameter ID="CntValueMax" Caption="Max CNT value (%)"
Type="Integer">80</Parameter>
      <Parameter ID="AngMin" Caption="Angle of min CNT (deg)"
Type="Integer">90</Parameter>
      <Parameter ID="AngMax" Caption="Angle of max CNT (deg)"
Type="Integer">140</Parameter>
    </Group>
    <Group ID="Format" Caption="Format">
      <Parameter ID="ExtAxesGroup" Caption="External axes group" Type="Integer">
        <Value Caption="Group 1" Default="true">1</Value>
        <Value Caption="Group 2">2</Value>
      </Parameter>
      <Parameter ID="RobotHolds" Caption="Robot holds" Type="Integer">
        <Value Caption="Tool" Default="true">0</Value>
        <Value Caption="Part">1</Value>
      </Parameter>
      <Parameter ID="UFrameFormat" Caption="UFRAME format" Type="Integer">
        <Value Caption="UFRAME_NUM=1" Default="true">0</Value>
        <Value Caption="UFRAME[1]=XYZABC">1</Value>
      </Parameter>
      <Parameter ID="UToolFormat" Caption="UTOOL format" Type="Integer">
        <Value Caption="UTOOL_NUM=1" Default="true">0</Value>
        <Value Caption="UTOOL[1]=XYZABC">1</Value>
      </Parameter>
    </Group>
  </UserDefinedParameters>
</Settings>
```

Of course, you can edit this file manually inside Visual Studio Code. The PostSchema.xsd file even contains the syntax rules which will give you error highlighting and predefined words autocompletion for

this purpose. But inside CLData Viewer we have a window which is specially designed to edit the settings.

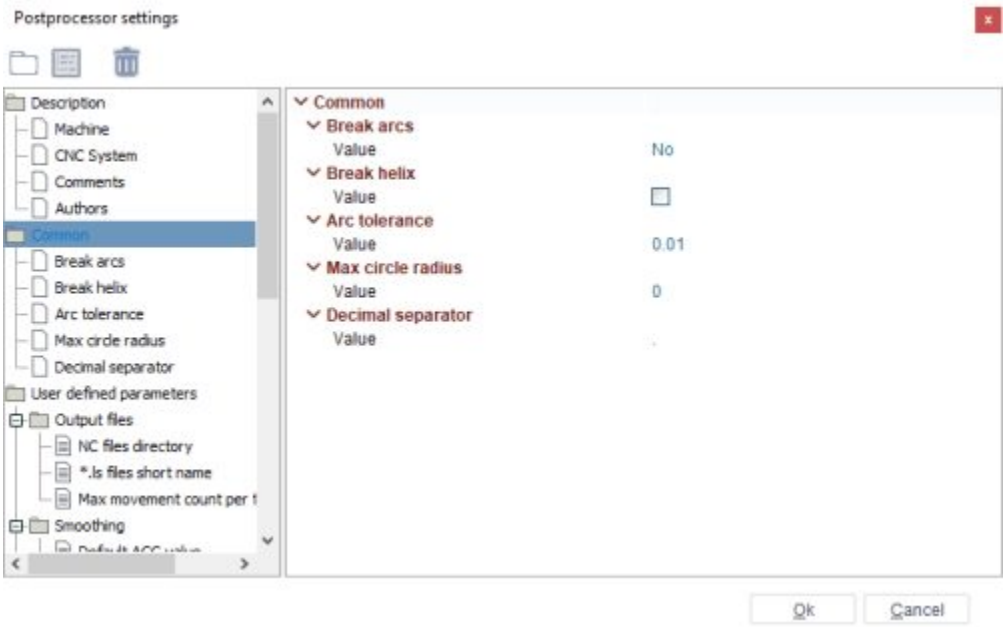
Postprocessor settings window

The postprocessor settings window allows you to work with the xml settings of a postprocessor in the most convenient way. In order to open the window, you need to select a postprocessor in CLData Viewer and right-click on it. After that click on Edit postprocessor XML settings in the context menu.



The postprocessor settings window displays parameters in the form of a visual tree of nodes and the properties inspector.

Postprocessor settings window:



Control elements

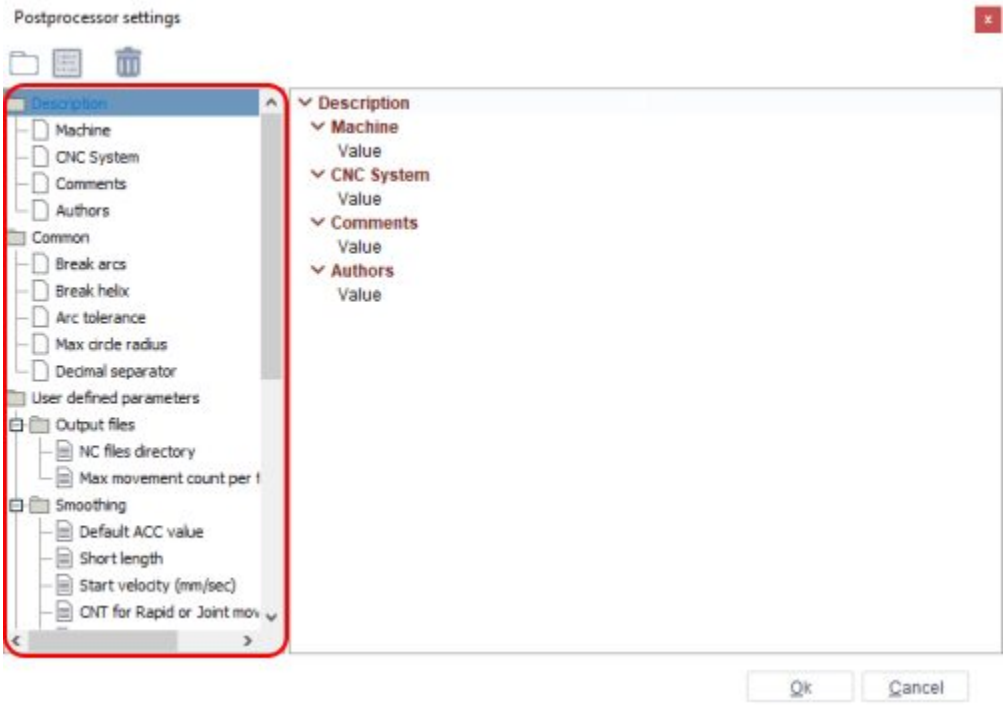
The tree element control panel is in the upper left part of the window. There are 3 controls on the panel:



1. Add group — adds a user group to the selected folder of the User defined parameters section and in its root;
2. Add parameter or enum value — adds a user parameter to the selected folder of the User defined parameters section and in its root. There are 7 variants to add a parameter:
    1. String parameter;
    2. Integer parameter;
    3. Double parameter;
    4. Boolean parameter;
    5. FileName parameter;
    6. FolderName parameter;
    7. Enum value — this value is added only to the selected parameter of User defined parameters.
3. Delete item — deletes the user-selected item in the elements tree.

Elements of tree

On the left part of the window there is a tree with default parameters and user parameters.



Tree display:

1. Section/folder — ;
2. Default parameter — ;
3. User parameter — ;
4. Enum value — ;

Default parameters

The Description section and Common section are default and parameters within them cannot be edited, added or deleted. It is only allowed to enter the parameter value. The Description section stores string parameters to describe the postprocessor. It has 4 parameters:

1. Machine;
2. CNC System;
3. Comments;
4. Authors.

The Common section stores parameters to set the default settings for the postprocessor. It has 5 parameters:

1. Break arcs — allows you to set the breaking arcs mode. It has the following values:
    1. No — without breaking;
    2. Cuts — with breaking;
    3. Halves — breaking an arc in half;
    4. Quarters — breaking an arc in quarters;
2. Break helix — allows you to enable or disable the breaking helix mode;

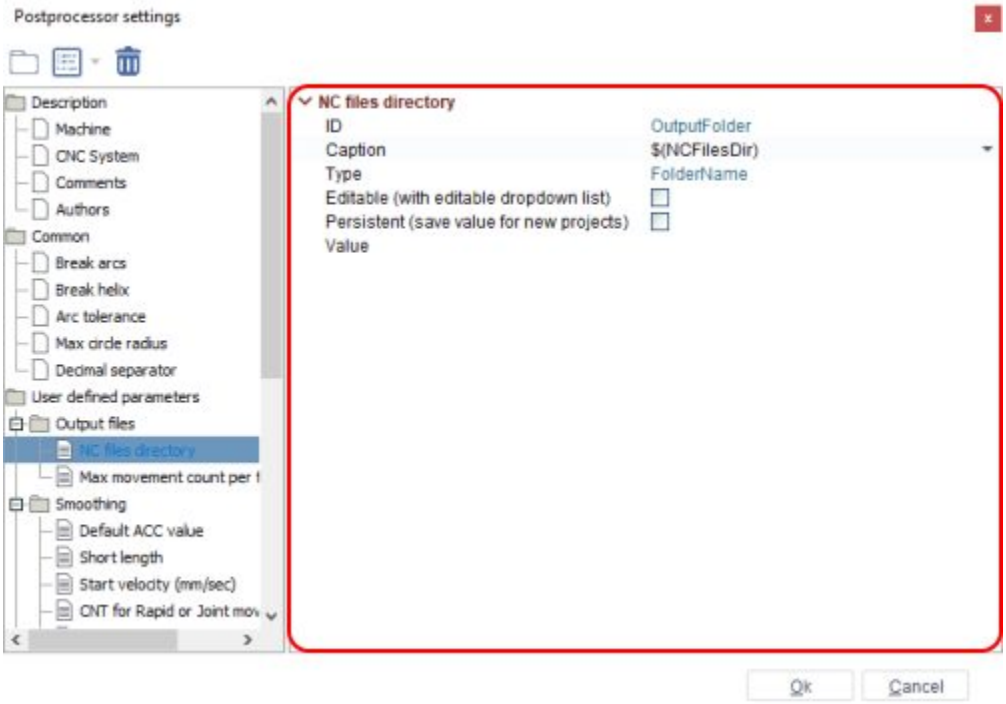
3. Arc tolerance — setting arc tolerance as a double value;
4. Max circle radius — setting the maximum circle radius as a double value;
5. Decimal separator — available values: "." and ",".

User parameters

The user can create their own parameters and store them in their own folders in the User defined parameters section. Folders can be stored in other folders. You can use drag-n-drop to move parameters and folders but an Enum value can be moved only in the current parameter.

Attributes of elements panel

The attributes of elements panel is on the right side of the window.



In the case of default parameters, you can only enter their values in this panel. In the case of user parameters, you can do full customization.

Each user element has a Caption attribute. You can set your own caption but there are several reserved variables to specify the caption:

1. $(OutputFiles);
2. $(NCFileExt);
3. $(NCFileName);
4. $(NCFilesDir);
5. $(NCProgNumber). These variables allow you to make a translatable name (localization). Also they allow generating some default values depending on the meaning of the parameter, for example, NC files are stored by default in the folder specified in the CAM system settings.

A user folder element has an ID attribute: a unique identifier of the element.

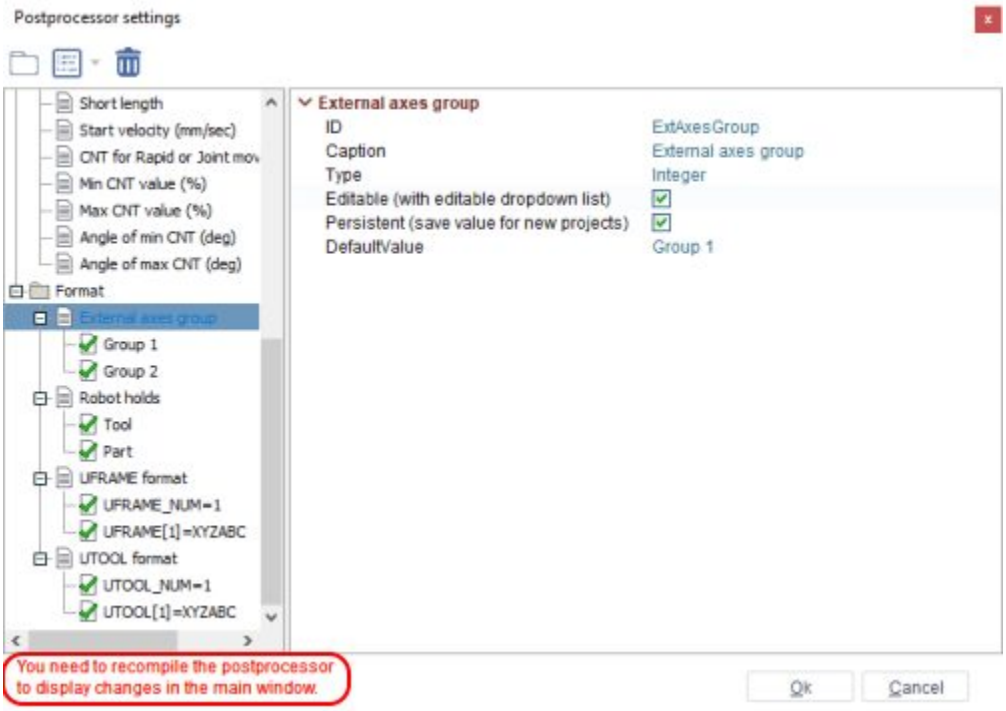
A user parameter element has the following attributes:

1. ID — unique identifier of the element;
2. Type — type of the parameter. There are several values:
    1. String — parameter with a string value;
    2. Integer — parameter with an integer value;
    3. Double — parameter with a double value;
    4. Boolean — parameter with a boolean value, check box;
    5. FileName — parameter with a dialog box for searching and specifying the path of files in the parameter value;
    6. FolderName — parameter with a dialog box for searching and specifying the path of a folder in the parameter value.
3. FilesFilter — file extension to search and open in the dialog box. Only for Type=FileName;
4. Editable (with editable dropdown list) — allows you to edit an existing enum value and add a new value;
5. Persistent — saves the value for new projects;
6. Value — parameter value according to its type;
7. DefaultValue — default value for a parameter that has enumerated values, dropdown list.

An enum value element has a Value attribute. The type of value is the same as the parent parameter.

Saving changes

After each change and move of a parameter, a warning will appear in the lower left part of the window: "You need to recompile the postprocessor to display changes in the main window".



In order for the changes to be saved in the xml file, you need to click the Ok button.

The main window of CLData Viewer shows the settings which are embedded into the dll file of the postprocessor, but not the source xml file. So you should rebuild the postprocessor in Visual Studio Code to see the result of editing in the main window.

It is also sometimes useful to reset the settings to the default state because CLData Viewer also stores the values you specify in the main window for debug purposes in a separate temporary file. To do this just use the Reset postprocessor settings to default popup menu item of the postprocessor or properties inspector.

Accessing settings from C# code

Below is an example of how to get access to the settings from the C# code of a postprocessor using the Settings and Settings.Params properties of the Postprocessor class:

```
void StartNewFile() {
    ls = new LSFile();
    ls.Name = Settings.Params.Str["OutFiles.LSFileName"];
    ls.OutputFileName = Settings.Params.Str["OutFiles.OutputFolder"] + @"\" +
ls.Name + ".ls";

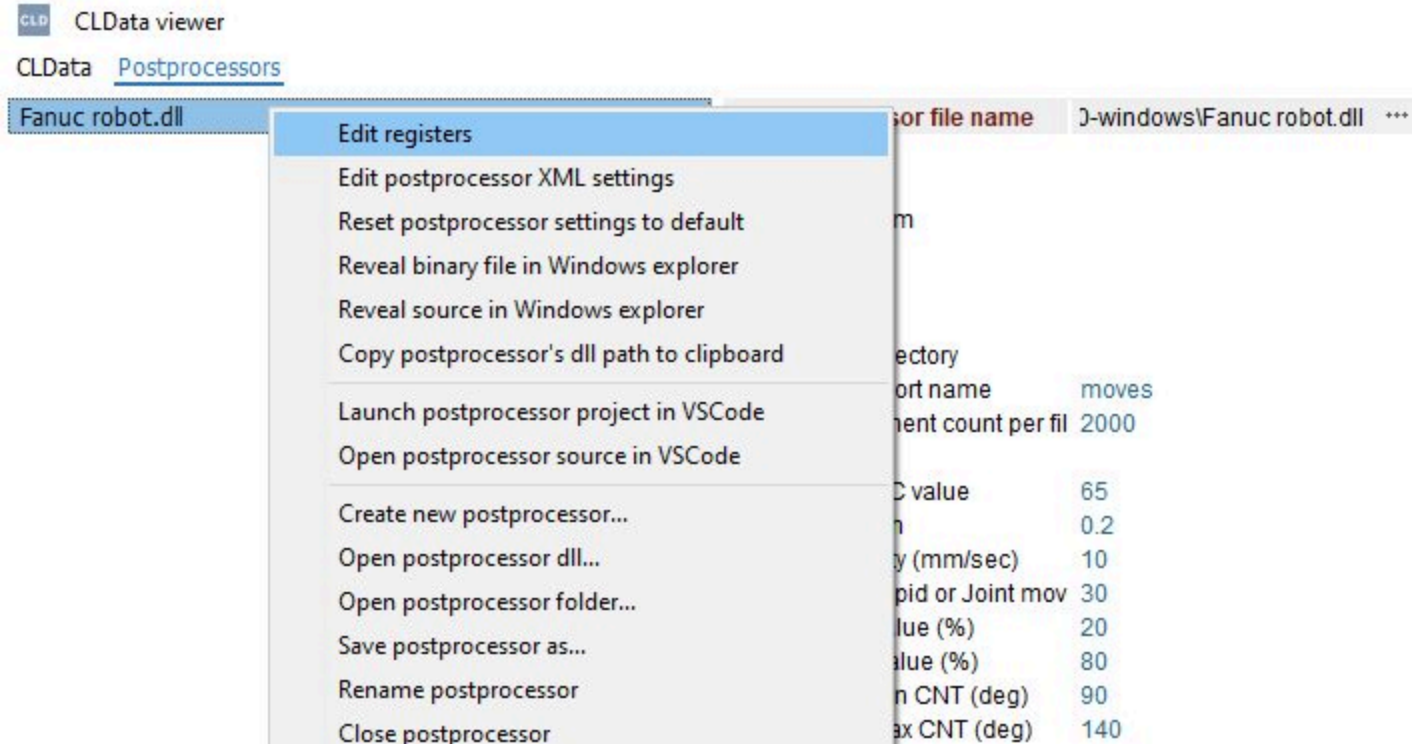
    unitsAreInches = prj.Int["Units"]>0;
    MaxMoveCount = Settings.Params.Int["OutFiles.MaxMoveCount"];

    smooth.accDefaultValue = Settings.Params.Int["Smoothing.DefaultACC"];
    smooth.ShortLength = Settings.Params.Flt["Smoothing.ShortLength"];
```

```
    smooth.StartVelocity = 60*Settings.Params.Flt["Smoothing.StartVelocity"];
    smooth.cntValueRapid = Settings.Params.Int["Smoothing.CntValueRapid"];
    smooth.cntValueMin = Settings.Params.Int["Smoothing.CntValueMin"];
    smooth.cntValueMax = Settings.Params.Int["Smoothing.CntValueMax"];
    smooth.angMin = Settings.Params.Int["Smoothing.AngMin"];
    smooth.angMax = Settings.Params.Int["Smoothing.AngMax"];
}
```

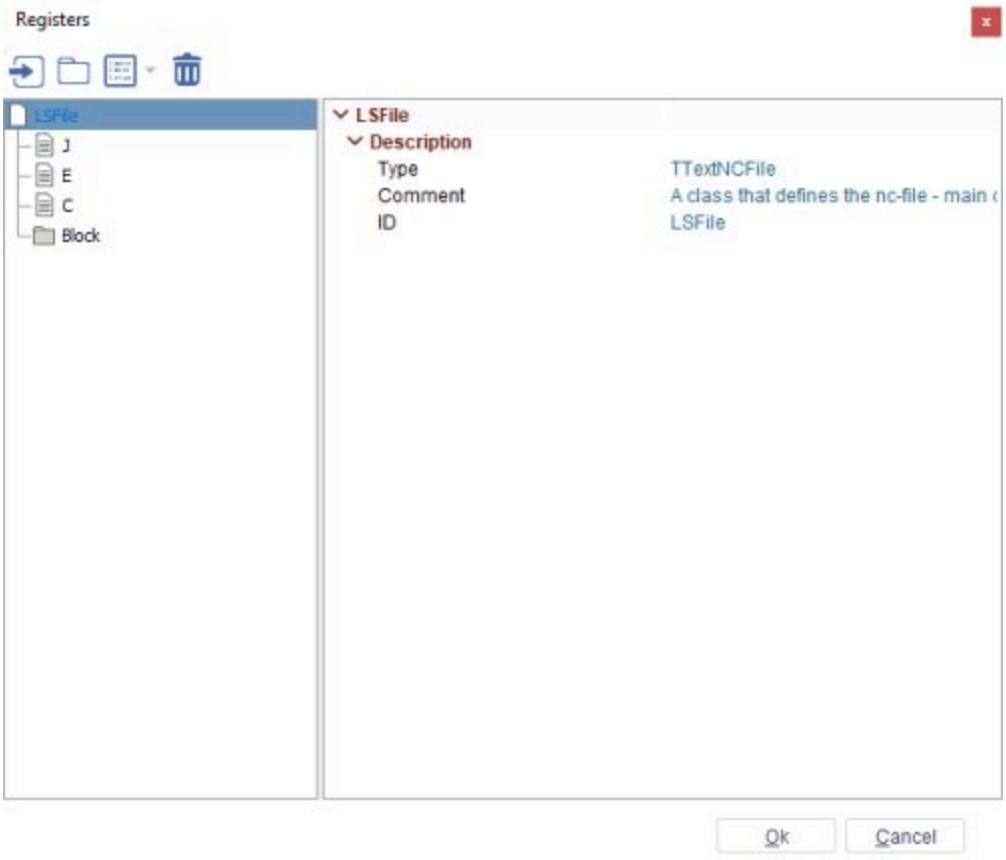
Registers window

The registers window allows you to work with registers. In order to open the window, you need to select a postprocessor in CLData Viewer and right-click on it. After that click on Edit registers in the context menu.



The registers window displays registers, blocks and files that are stored in the Registers.cs file of the postprocessor. The window allows you to edit registers, blocks and files more conveniently.

Registers window:



Registers.cs file:

```
// This file may be autogenerated, do not place meaningful code here.
// Use it only to define a list of nc-words (registers) that may appear
// in blocks of the nc-file.
namespace DotnetPostprocessing.SDK
{
    ///<summary>A class that defines the nc-file - main output file that should be
generated by the postprocessor.</summary>
    public partial class LSFile: TTextNCFile
    {
        ///<summary>The block of the nc-file is an ordered list of
nc-words</summary>
        public NCBlock Block;

        ///<summary>Automatic block numbers counter</summary>
        // public CountingNCWord BlockN = new CountingNCWord("N{######}", 1, 1, 1);

        ///<summary>Joint value formatter</summary>
        public NumericNCWord J = new NumericNCWord("{-000.000}", 0);

        ///<summary>External axis value formatter</summary>
        public NumericNCWord E = new NumericNCWord("{-####.000}", 0);
```

```
        ///<summary>X, Y, Z, W, P, R value formatter</summary>
        public NumericNCWord C = new NumericNCWord("{-000!000}", 0);

        public LSFile(): base()
        {
            Block = new NCBlock(this);
            OnInit();
        }

    }
}
```

Control elements

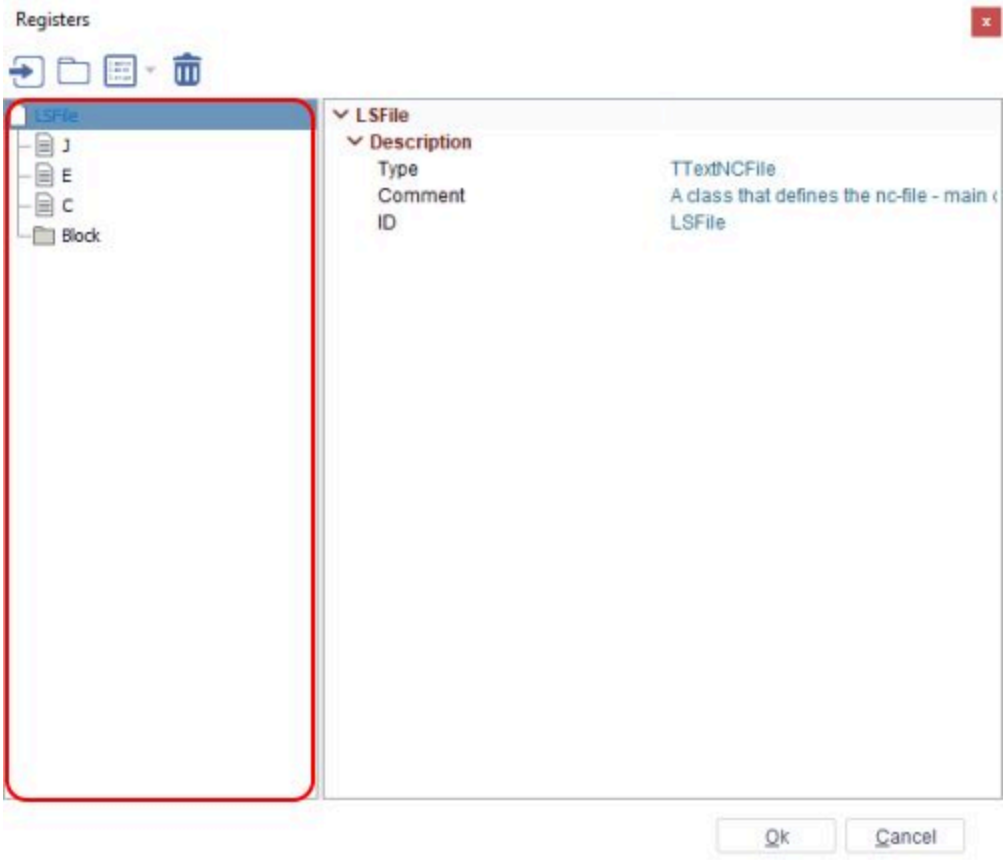
The tree element control panel is in the upper left part of the window. There are 4 controls on the panel:



1. Add file — adds a new output file class;
2. Add block — adds a new block that can store registers;
3. Add register — adds a user register to the selected block or file. There are 4 variants to add a register:
    1. Numeric register — register that stores a numeric value;
    2. Counting register — register that stores a numeric value that can be incremented;
    3. Text register — register that stores a string value;
    4. Registers of current file. This variant allows you to move an existing register to the selected item (file or block). If a register is moved to a file then it is removed from the block in which it was located. If a register is moved to a block then it is removed from the file in which it was located. But if it was in another block, then the register is moved to the selected block without deletion.
4. Delete item — deletes the selected item in the elements tree.

Elements of tree

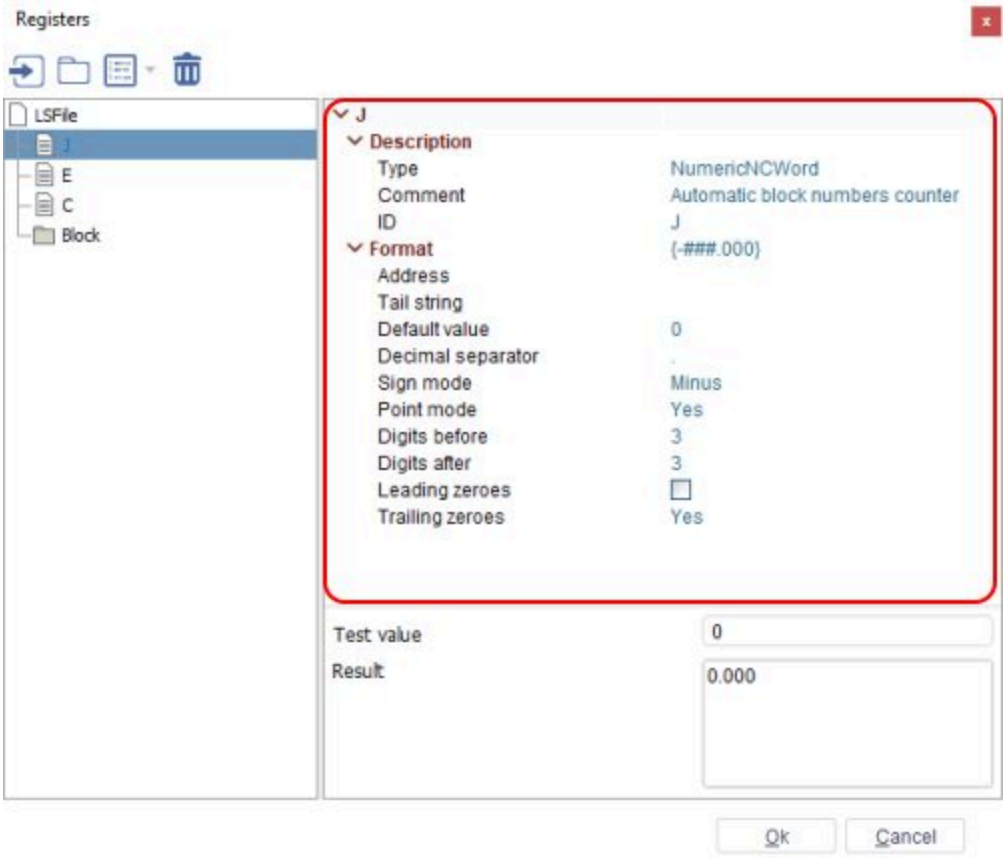
On the left part of the window there is a tree with a list of registers , blocks and files .



Registers can be located in a file or a block. You can move registers between blocks in a file. A register is moved by Drag'n'Drop. A register can be located either only in the root of the file or in one or more blocks of the given file. You need to move with the Ctrl button pressed to copy a register from one block to another. Each element of the tree has its own parameters.

Parameters of elements panel

The parameters of elements panel is on the right side of the window.



There are different parameters for each type of element.

Parameters of file element

A file element has the following parameters:

1. Type — type of the output file class:
    1. TTextNCFile — the main class to use for writing text files;
    2. TSimpleTextNCFile — simplest implementation of the output file;
2. Comment — hint comment for the file;
3. ID — unique name of the file class.

Parameters of block element

A block element has the following parameters:

1. Comment — hint comment for the block;
2. ID — unique name of the block class.

Parameters of register element

A register element has the following parameters:

1. Type — type of the register class:

    1. CountingNCWord — it's a NumericNCWord that automatically increments or decrements the value of the register with the defined step after each call of Out() or Form();
    2. NumericNCWord — type that converts numeric values to a text format;
    3. TextNCWord — type that converts string values to a text format;
2. Comment — hint comment for the register;
3. ID — unique name of the register class; A register element has an additional section. It's the format. You can enter the output format in the right part of the format line using reserved characters. After entering parameters, parameters inside the format section are changed in accordance with the entered format string inside. And vice versa, when you change parameters inside the format section, the format string changes. Each type of register class has its own parameters in the format section.

Format parameters of NumericNCWord register

The NumericNCWord register has its own parameters:

1. Address — the symbols which will be written in the NC-program block before the register value;
2. Tail string — the symbols which will be written in the NC-program block after the register value;
3. Default value — default value of the current register;
4. Decimal separator — using "." or "," as the decimal separator;
5. Sign mode — defines the output mode for the sign of the register value. The following options are available:
    1. No — sign is not output;
    2. Minus — only minus is output;
    3. Plus and minus — plus and minus are output;
6. Point mode — output decimal separator. This field can have the following values:
    1. No — decimal separator is not output;
    2. Yes — decimal separator will be output if the register value has a fractional part;
    3. Always — decimal separator will be output anyway;
7. Digits before — maximum quantity of signs in the whole part of the number;
8. Digits after — maximum quantity of signs in the fractional part of the number;
9. Leading zeroes — defines the zeroes output mode before the register value;
10. Trailing zeroes — defines the zeroes output mode after the register value.

Format parameters of CountingNCWord register

The CountingNCWord register has the same parameters as the NumericNCWord register but has its own additional parameters:

1. Auto increment step — the number by which the output value of the register will be increased;
2. Frequence — output the value of the register with the specified frequency. For example output every third value (Frequence = 3);
3. Preserve indent — aligns all numeric values by the value with the most count of characters.

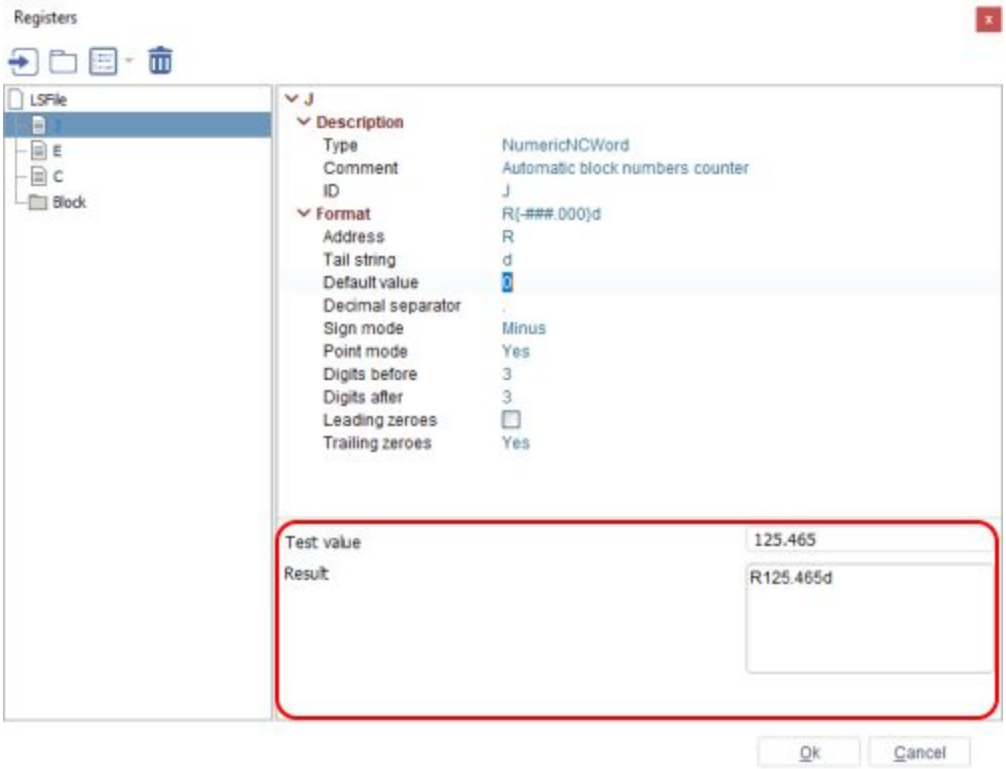
Format parameters of TextNCWord register

The TextNCWord register has its own parameters:

1. Address — the symbols which will be written in the NC-program block before the register value;
2. Tail string — the symbols which will be written in the NC-program block after the register value;
3. Default value — default value of the current register;
4. Text case — case of the register value. This parameter has three values:
    1. Original — output value in its initial state;
    2. Lower — output value in lower case;
    3. Upper — output value in upper case.

Test panel

The test panel is located below the parameters of elements panel. The test panel is displayed only for register elements.

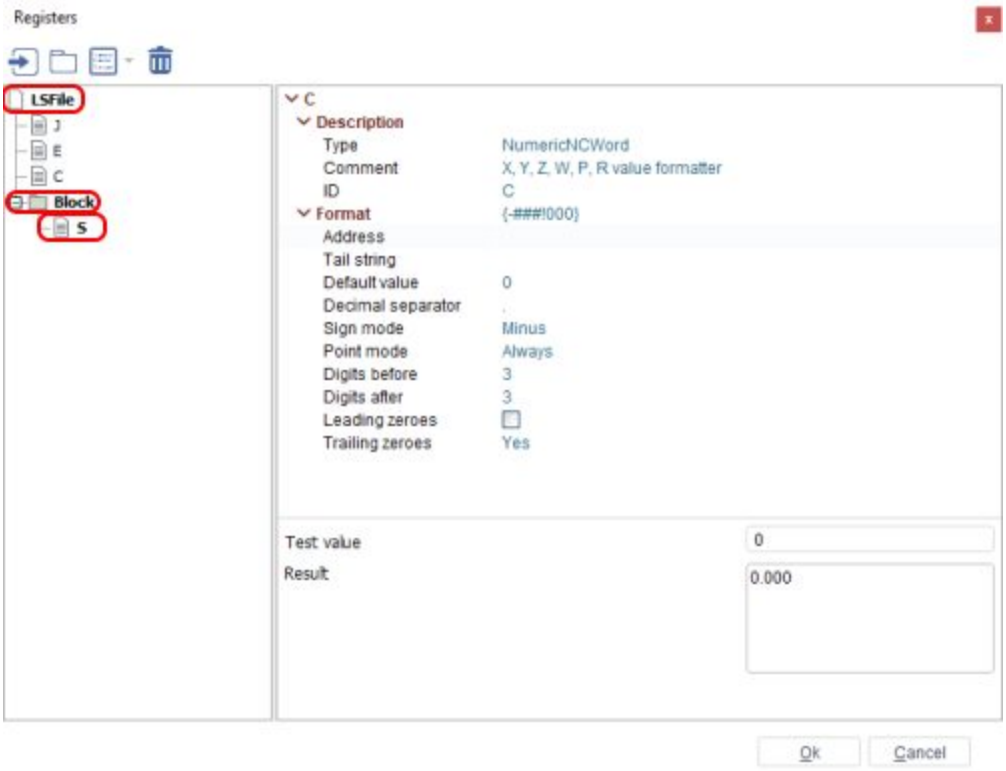


There are 2 fields on this panel:

1. Test value — value that will be output based on the specified format;
2. Result — example of displaying the entered value based on the specified format. Five values are displayed for the register, since this register is incremented with each value.

Saving changes

Each change, addition, removal and movement of elements is displayed on the panel. This element and its parents will be bolded.



In order to save all the entered changes in the register window, you need to click the Ok button. After that the file will be rebuilt and all data from the window will be entered into this file.

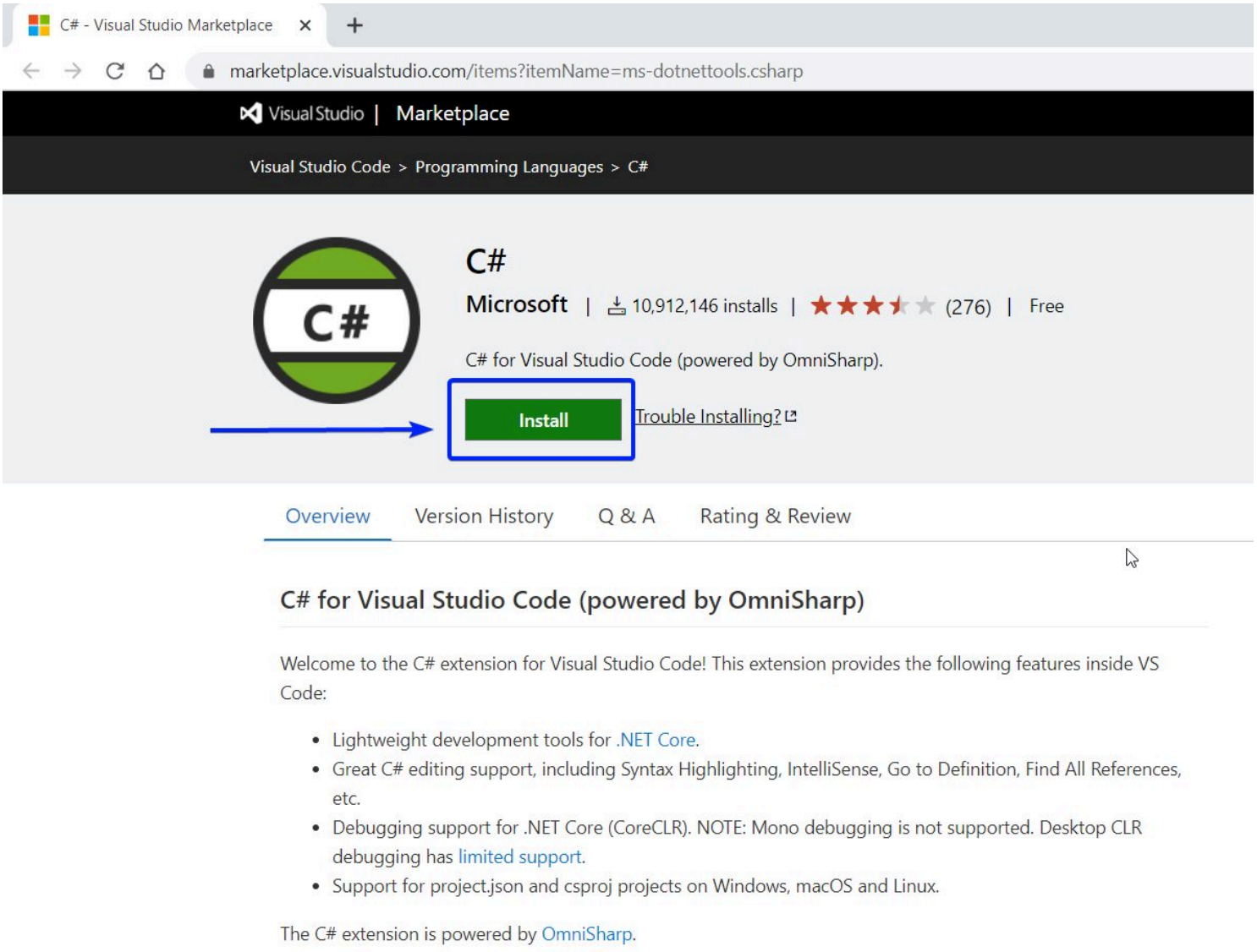
```
// This file may be autogenerated, do not place meaningful code here.
// Use it only to define a list of nc-words (registers) that may appear
// in blocks of the nc-file.
namespace DotnetPostprocessing.SDK
{
    ///<summary>A class that defines the nc-file - main output file that should be
generated by the postprocessor.</summary>
    public partial class LSFile : TTextNCFile
    {
        ///<summary>The block of the nc-file is an ordered list of
nc-words</summary>
        public NCBlock Block;
        ///<summary></summary>
        public NumericNCWord S = new NumericNCWord("S{-###.###}", 0);
        ///<summary>Automatic block numbers counter</summary>
        public NumericNCWord J = new NumericNCWord("{-###.000}", 0);
        ///<summary>External axis value formatter</summary>
        public NumericNCWord E = new NumericNCWord("{-####.000}", 0);
        ///<summary>X, Y, Z, W, P, R value formatter</summary>
        public NumericNCWord C = new NumericNCWord("{-###!000}", 0);
        public LSFile(): base()
        {
            Block = new NCBlock(
```

```
                this,
                S);
            OnInit();
        }
    }
}
```

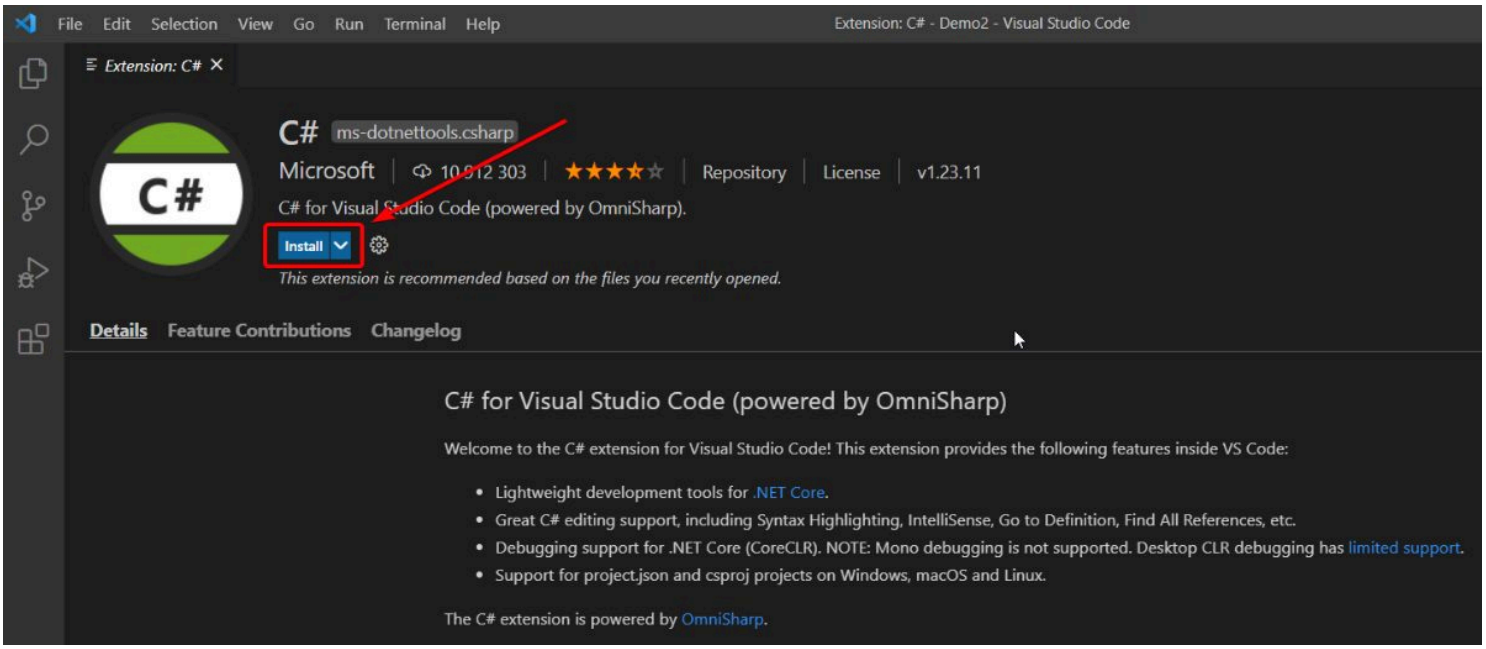
Prepare computer to develop .NET postprocessors

To be able to build the binary file *.dll of a postprocessor from its source files (Postprocessor.cs, Registers.cs, *.csproj etc.) follow the instructions below.

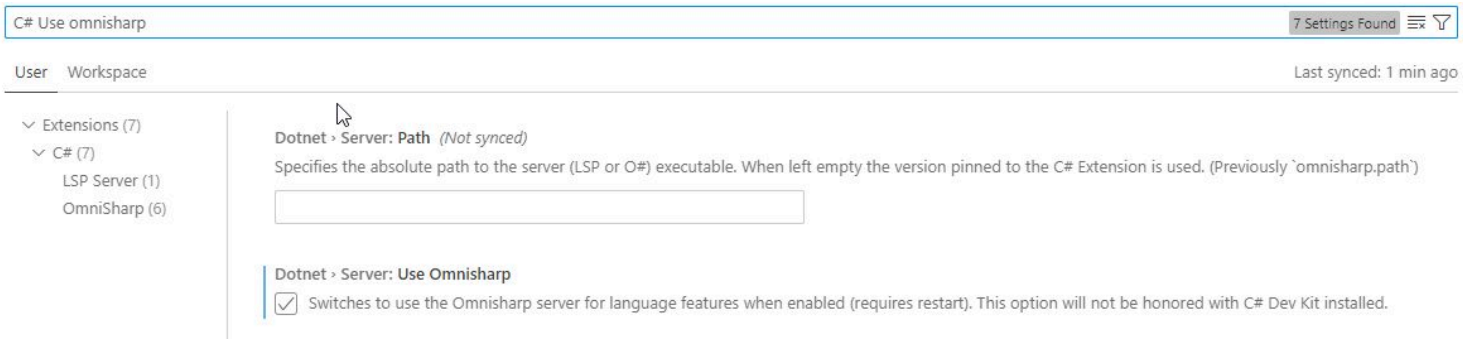
1. Download and install .NET SDK 8.0 x64 from the official site (https://dotnet.microsoft.com/download, size ~200 Mb). There is a simple step-by-step guide there.
2. To be able to write source files, download and install the latest available version of Visual Studio Code (https://code.visualstudio.com/download, size ~100 Mb).
3. Install the C# extension from the Visual Studio Code marketplace.



4. It will start a Visual Studio Code (abbreviated VSCode) instance on the Extensions tab. Just click the "Install" button next to the extension name.

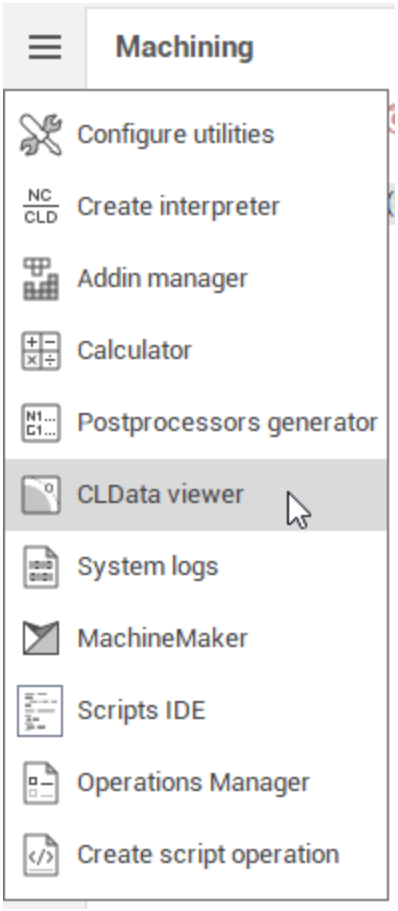


In the latest versions of VS Code, when installing an extension for `C#`, you are prompted to immediately install the "C# Dev Kit". However, according to the terms, this set requires a Visual Studio license. Using the "C# Dev Kit" extension is not necessary for developing postprocessors; everything works fine without it, and it does not require purchasing Visual Studio.

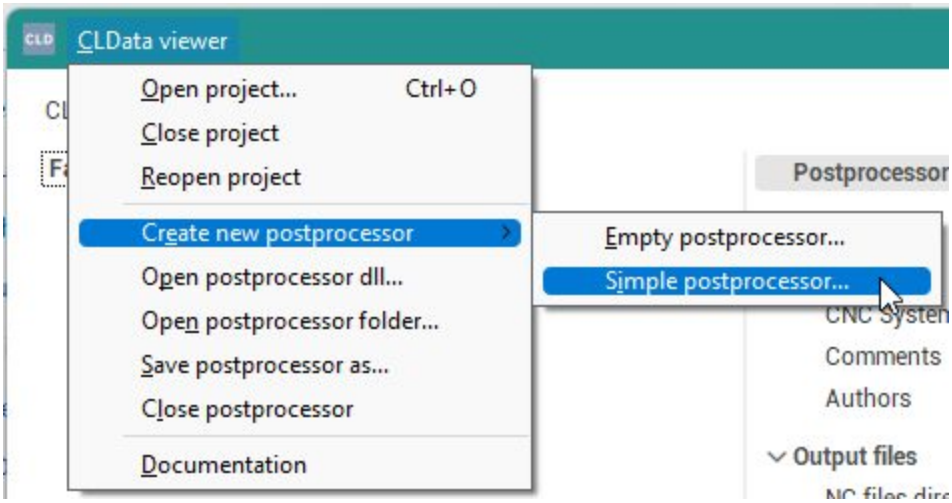


Sometimes the Omnisharp versions of the `C#` extension work better so you can try enabling the "C#: Use Omnisharp" setting which you can find after clicking on the gear button next to the extension install button.

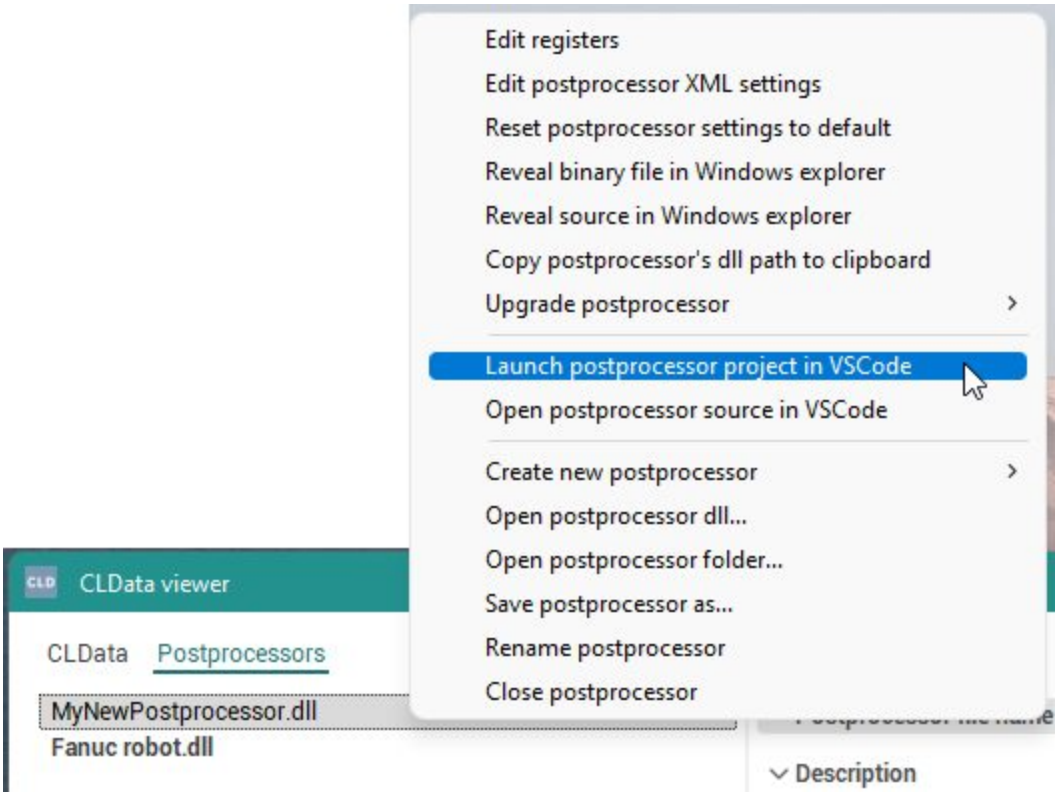
5. It is recommended (but not required) to install the extension for editing xml files "XML Language Support by Red Hat" (https://marketplace.visualstudio.com/items?itemName=redhat.vscode-xml).
6. If you have not yet, download and install the latest version of the CAM system from the official resource.
7. Start the CAM system and launch the CLData Viewer from the Utilities menu.



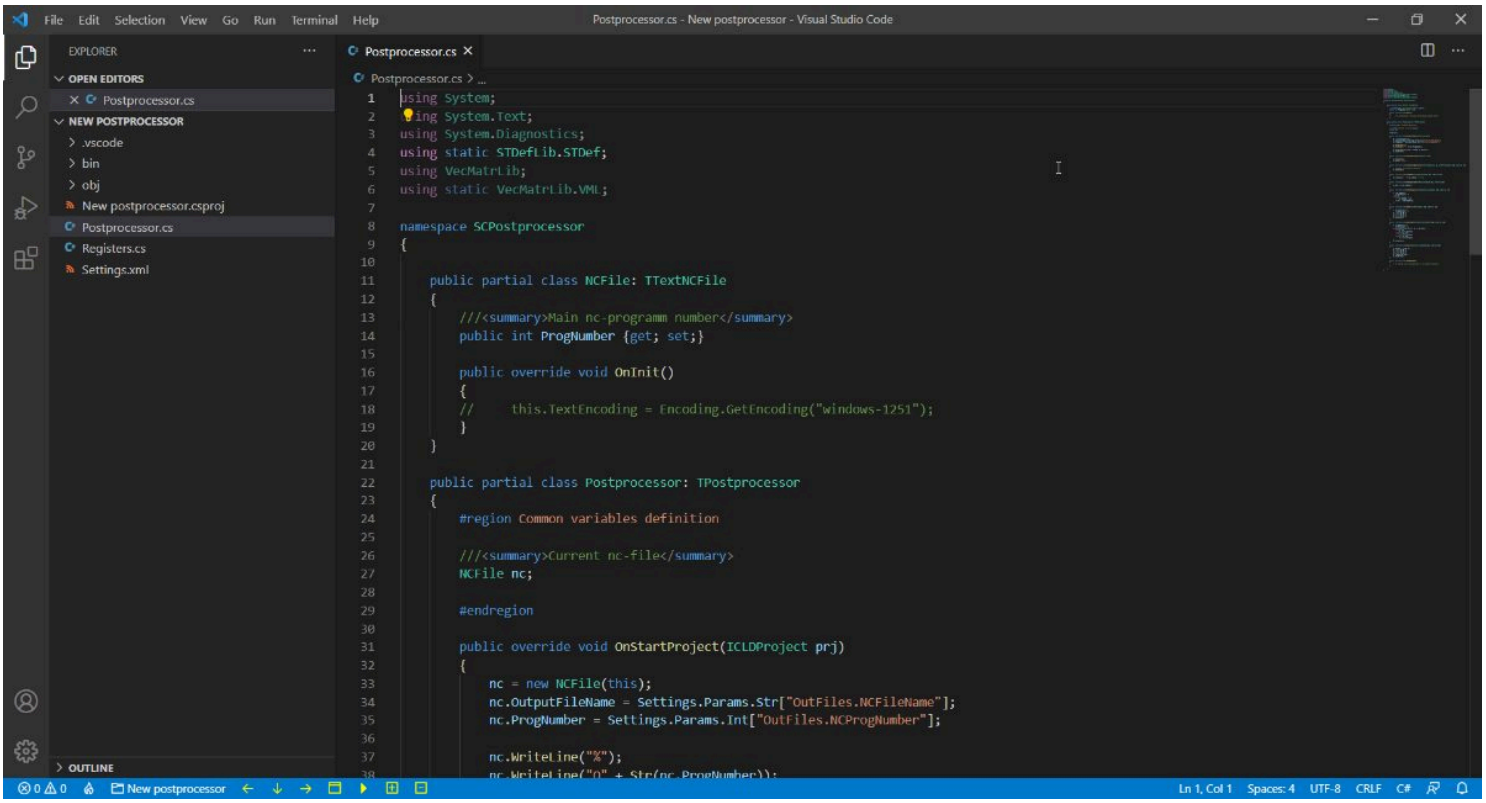
8. It will add the "InpCoreDir" environment variable to the Windows' variables. To be able to launch Visual Studio Code from the CAM system, the system "Path" environment variable should include the VS Code installation folder. Usually the VSCode installer adds everything itself and you just need to restart the computer for the environment variables to take effect. We recommend restarting the computer after the first launch of the CLData Viewer for environment variables to take effect.
9. Start the CAM system and launch the CLData Viewer again. Now you can try to create a new postprocessor. Click the main menu CLData viewer/Create new postprocessor/Simple postprocessor….



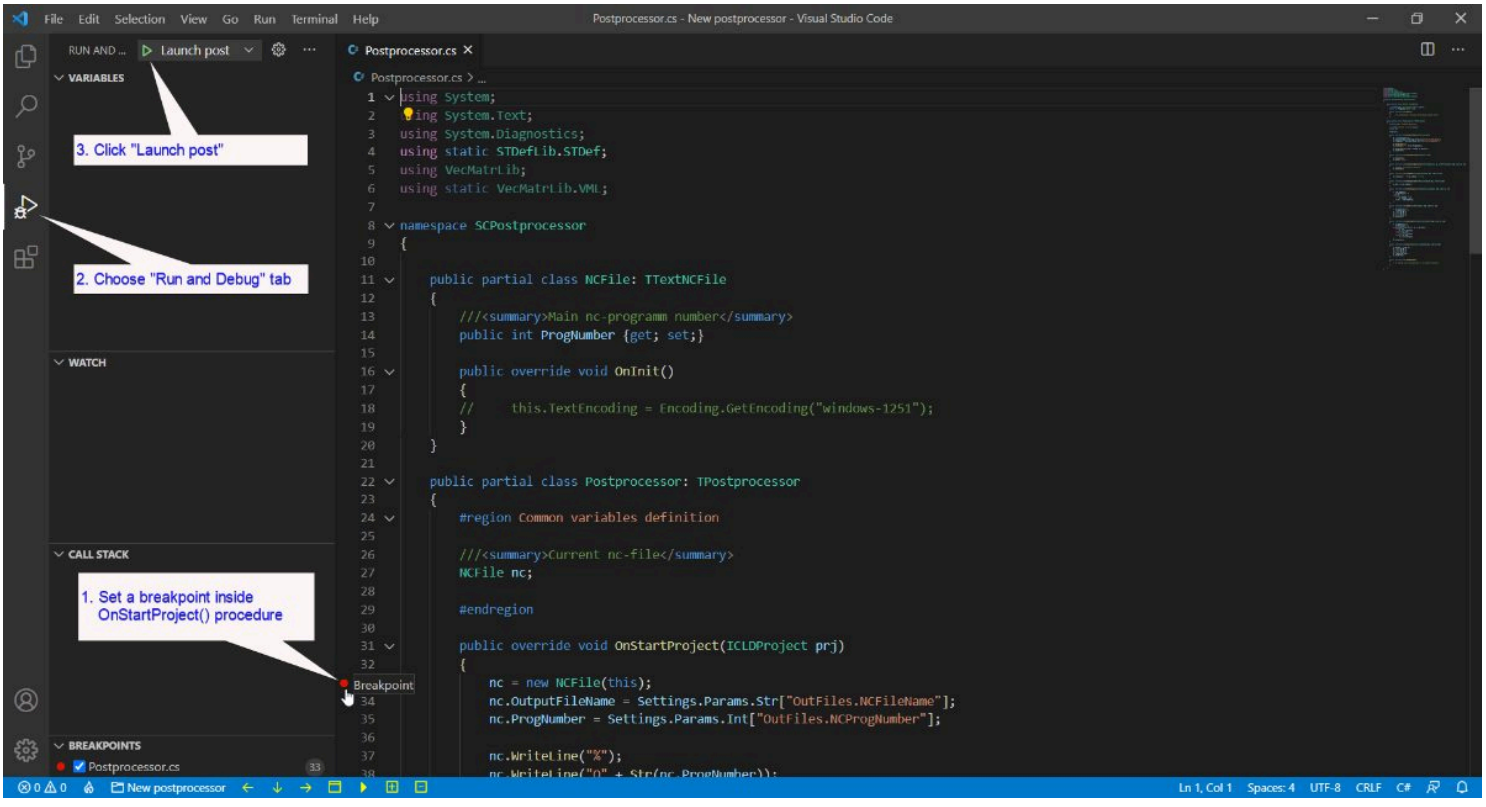
10. A standard folder selection dialog box will open. Here you will need to create a new empty folder that will contain all the postprocessor files. The folder name will match the name of the new postprocessor.
11. In the context menu of the postprocessor choose the "Launch postprocessor project in VS Code" item or just double-click on the postprocessor in the list.



12. The new VSCode window will appear with the main Postprocessor.cs source file of the selected postprocessor. The first time you open a postprocessor, it may take some extra time for the Omnisharp extension (C#) to download the latest updates and dependencies automatically.

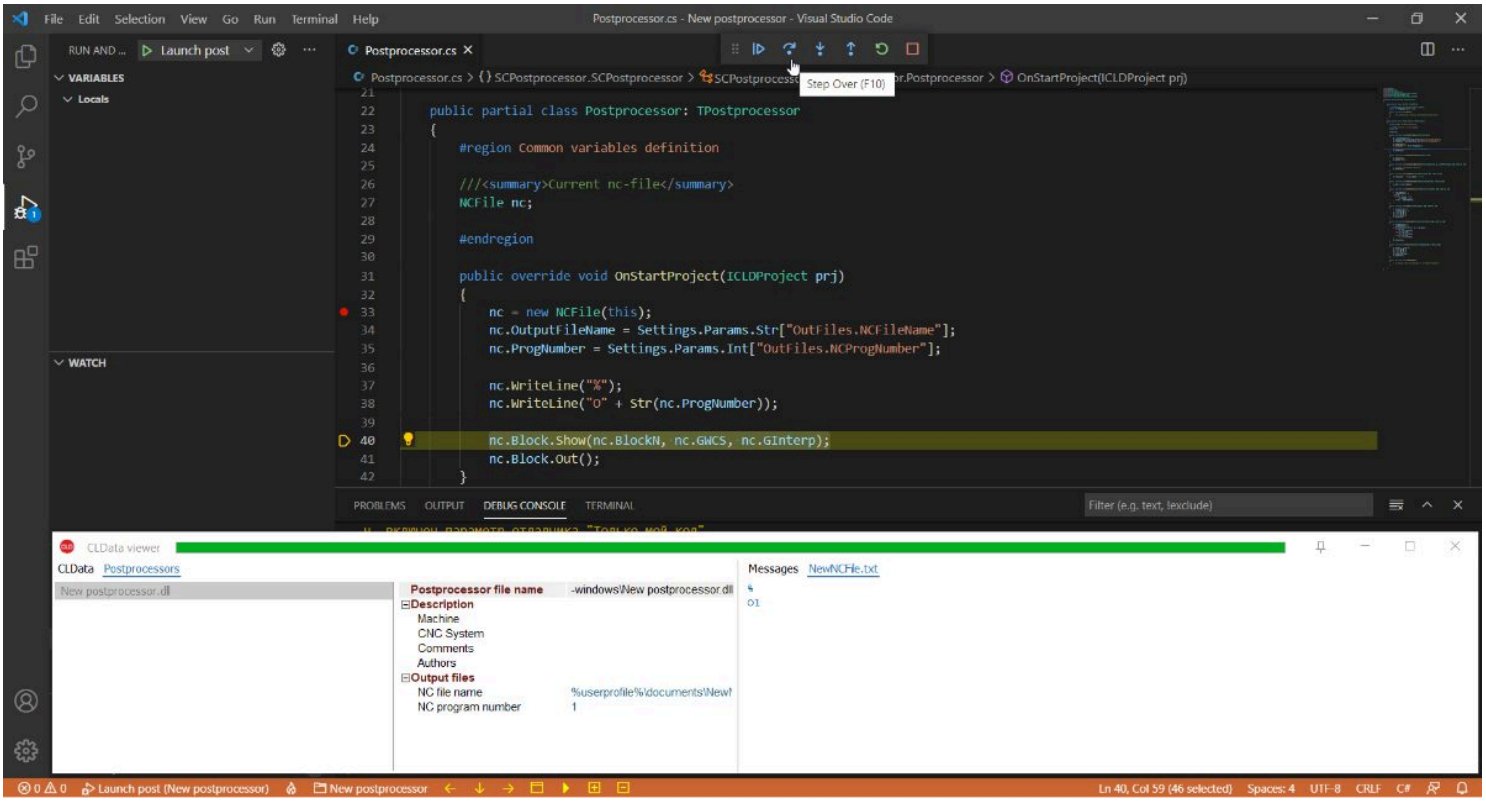


13. Then you can set a breakpoint inside the OnStartProject procedure, choose the "Run and Debug" tab and click "Launch post" (F5 key).

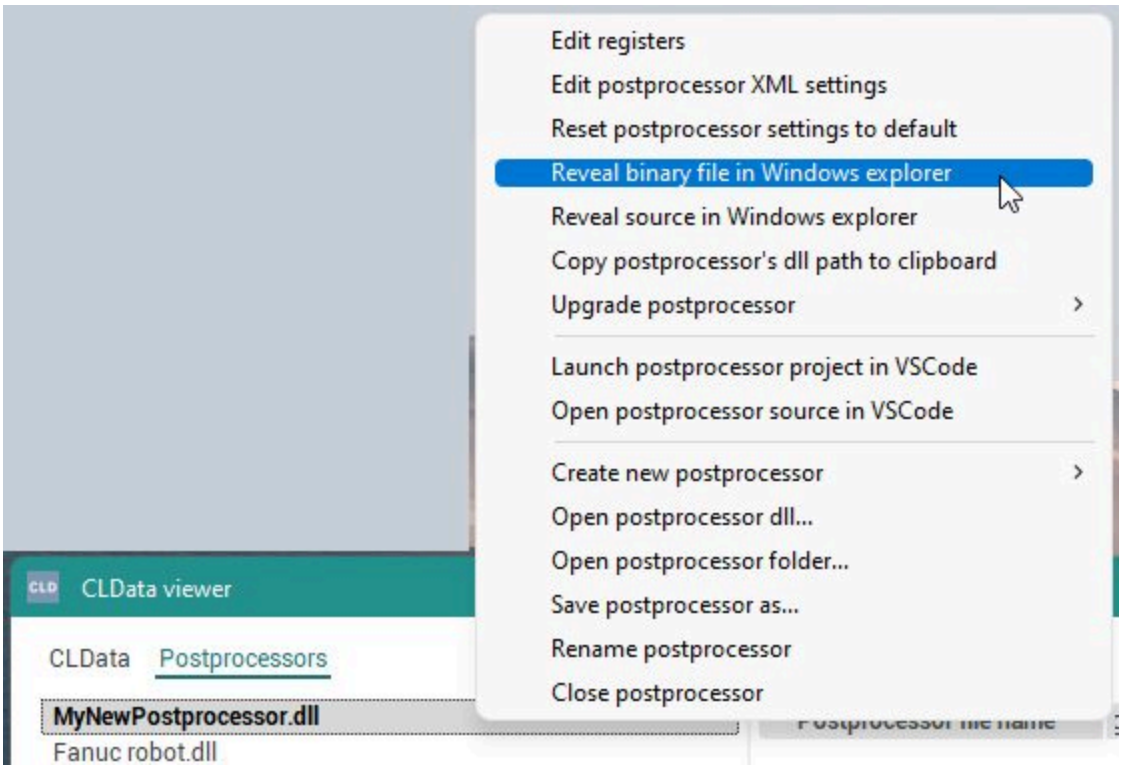


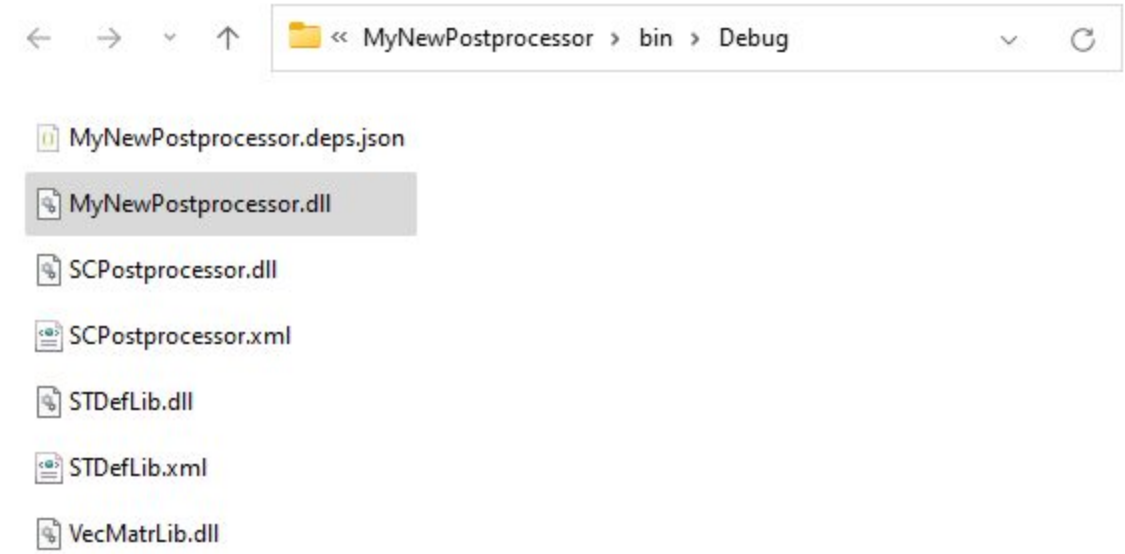
14. It will build the postprocessor from sources and start debugging. The CLData Viewer window should appear (if it was minimized before). Execution should stop at the defined breakpoint. Now you can

use Step Over (F10) at the toolbar in order to walk through the lines of the postprocessor code. As the lines are written to the output file, the CLData Viewer will show intermediate results.



15. Finally, you can modify the postprocessor code in the Postprocessor.cs, Registers.cs and Settings.xml files as you need and debug it using all available features of Visual Studio Code. If you are not yet familiar with the capabilities of VSCode, we recommend that you visit the "Get started" section from its official documentation (https://code.visualstudio.com/docs/getstarted/introvideos).
16. After you have written and debugged the postprocessor source code, you can obtain and use the postprocessor binary output *.dll file. Just select "Reveal binary file in Windows explorer" from the context menu of the postprocessor in the CLData Viewer. This will select the postprocessor binary in explorer from where you can copy or send it to the customer.


Edit registers
Edit postprocessor XML settings
Reset postprocessor settings to default
Reveal binary file in Windows explorer
Reveal source in Windows explorer
Copy postprocessor's dll path to clipboard
Upgrade postprocessor
Launch postprocessor project in VSCode
Open postprocessor source in VSCode
Create new postprocessor
Open postprocessor dll...
Open postprocessor folder...
Save postprocessor as...
Rename postprocessor
Close postprocessor
CLData viewer
CLData
Postprocessors
MyNewPostprocessor.dll
Fanuc robot.dll


MyNewPostprocessor › bin › Debug
MyNewPostprocessor.deps.json
MyNewPostprocessor.dll
SCPostprocessor.dll
SCPostprocessor.xml
STDefLib.dll
STDefLib.xml
VecMatrLib.dll

Develop .NET postprocessors

When to use

Work on a C# postprocessor: typed handlers, an SDK project, a compiled postprocessor assembly. Run `inspect-cldata` first and `verify-nc-program` afterwards.

There is no dedicated MCP server for .NET postprocessors. You work with the C# project through the ordinary file and build tools, and the postprocessor-specific parts through the DotNet Posts extension or the batch runner.

Choose the starting point

Adapting an existing postprocessor is the normal case; creating one from a template is the exception. Decide explicitly, and say which you chose and why.

1. Closest existing postprocessor — prefer this. A distributed postprocessor supplied with the CAM system, or one from the user's own corpus, for a similar machine and control system. Copy it, then work from its actual code: it already builds and produces output, so the baseline is real. Never carry over machine-specific behaviour without checking it against this project's data.
2. New from a template — only when nothing close exists. The user creates postprocessors in the CLData Viewer UI, which copies and renames a template for them. When you are asked to do it instead, copy a template from `Supplement/Postprocessor/DotNet/Templates` (`EmptyPost` or `SimplePost`) yourself, and keep the SDK package reference, the target framework and the project settings exactly as the template has them. Never choose a version from memory — the installed release decides it. Do not ask the user to copy and rename files by hand.

The renaming rule is hard: the folder name and the **`.csproj`** name must be identical. The configuration that comes with the template derives both from the folder name: the build task in `.vscode\tasks.json` builds `${workspaceFolder}\${workspaceFolderBasename}.csproj`, and the launch configuration in `.vscode\launch.json` hands the runner `bin\Debug\${workspaceFolderBasename}.dll`. Name the `.csproj` differently and both the `F5` build and the debugger stop working, complaining about a file that does not exist rather than about a name. So rename the folder and the `.csproj` in one move, do not set `AssemblyName` by hand (the assembly name follows the project name), and leave the `.vscode` folder as it is. The same rule applies when you copy an existing postprocessor into a folder with a new name.

Then, whichever way you started:

3. Build the project unchanged and record framework, warnings and output assembly. If it does not build as it is, stop and report that — nothing after it is diagnosable.
4. Generate an NC program from the unchanged project and keep it as the baseline.

Tools

DotNet Posts commands, available in VS Code and callable by an agent that can execute commands:

Command	Purpose
`dotnetPosts.status`	Configurations, paths and post input parameters; read-only, does not open or change the panel
`dotnetPosts.setup`	Prepare a run configuration without starting it
`dotnetPosts.run`	Run the configured generation
`dotnetPosts.cancel`	Cancel a running generation
`dotnetPosts.symbols` / `dotnetPosts.goto`	List the postprocessor handlers; open one
`dotnetPosts.highlight` / `dotnetPosts.whereAmI`	Mark a range of code; report where the user's cursor is
`dotnetPosts.inspectCldata`	Open the configured CLData in the Inspector
`dotnetPosts.generateNC` / `dotnetPosts.generateNCFromDll`	Generate from the workspace project, or from an assembly you point at

`dotnetPosts.autoBuild` builds the project before a run, which prevents testing an edit against a stale assembly. `dotnetPosts.installationFolder` must point at the CAM installation containing the batch runner.

The batch runner is `InpCore.exe`. It has two modes that matter here: writing the postprocessor's default settings out as a `Settings.xml` next to the input, which is what makes running an assembly without sources possible, and processing a CAM project in batch mode. It reports progress as a stream of events ending with a result code, and the process exit code repeats the outcome — a run counts as successful only when both agree. The settings format is not the SPPX one; use the file the runner produced. If the installed runner does not support extracting settings, report the version limitation instead of hand-writing a settings file.

Command lines

There is no MCP server for .NET postprocessors, so when DotNet Posts is unavailable you drive the same programs yourself. `InpCore.exe` lives in the CAM installation — the folder in `dotnetPosts.installationFolder`, or its `Bin64`. Quote any path that contains spaces.

Take that path from the setting or from what the user tells you, and name the executable you used in your report. There is also an `InpCoreDir` environment variable, but treat it as a hint, not as the answer: the CLData Viewer writes it when it starts, so on a computer with several CAM versions installed it holds

whichever installation's viewer ran last — which may not be the one the user is working with. The trap is quiet, because a run against the wrong version usually succeeds and merely produces different output.

Debugging from the editor is covered: the template's `F5` configuration is written in terms of `${env:InpCoreDir}`, but the DotNet Posts extension substitutes the path from `dotnetPosts.installationFolder` when the session starts, and warns once if the two disagree. The variable remains the fallback when the setting is empty. Remember that changing it does not reach programs that are already running — in practice it takes signing out of Windows and back in.

Build.

```
dotnet build "<folder>\<Name>.csproj"
```

The template's own build task adds `/property:GenerateFullPaths=true /consoleloggerparameters:NoSummary`, which makes the compiler report full paths and stops duplicate error summaries — use the same flags when you intend to parse the output.

Extract **`Settings.xml`** from a compiled postprocessor. Needed when you have the DLL but no sources; a project of your own already carries its settings file.

```
InpCore.exe -settingsdump -postfile:"<folder>\bin\Debug\<Name>.dll" -
settingsfile:"<folder>\Settings.xml"
```

A non-zero exit code, or a settings file that is not there afterwards, means the extraction failed — report the version limitation instead of writing a settings file by hand.

Generate an NC program.

```
InpCore.exe -batchmode -postfile:"<...>\<Name>.dll" -cldfile:"<project>.stcp" -
settingsfile:"<...>\Settings.xml"
```

Progress and results are written as JSONL next to the settings file, as `<settings file>.events.jsonl`: one JSON object per line, with `event`, and depending on it `percent`, `severity`, `text`, `path`, `success`, `resultCode` and `durationMs`. The `file` event carries the path of what was written; the closing `done` event carries `success` and `resultCode`. Read that file rather than the console output.

Two rules about the result. A run counts as successful only when `done.success` is true and the process exit code is 0; if events appeared but no `done` did, the runner stopped early and the NC file, if any, is not to be trusted. And the output path is not a command-line argument: the NC file name and directory are parameters of the settings file, so set them there before the run.

Where the human is meant to watch instead, the template's `F5` configuration runs `InpCore.exe -cldviewermode -sharedfile:STInpCoreOfDebugger -postfile:...` with the build task in front of it. That is the debugging path, not a batch run — do not mix the two.

Workflow

1. Get the command and its real parameter values from `inspect-cldata`, in the `dotnet` path dialect.
2. `dotnetPosts.symbols` to list the handlers, `dotnetPosts.goto` to open the relevant one. Read the current implementation before changing it.
3. Make one small change and state the expected NC output. Prefer named typed properties from the SDK over raw indexes wherever the SDK provides them.
4. Build. Read the warnings that concern your change.
5. `dotnetPosts.status` to see the configuration, then `dotnetPosts.run` on the reduced project. Capture the report: output file, messages, result code. Use `dotnetPosts.generateNCFromDll` when there are no sources, and `dotnetPosts.cancel` to stop a long run.
6. If DotNet Posts is unavailable, run `InpCore.exe` directly in batch mode with the settings file produced by the runner, and report that you used the fallback.
7. Compare the generated program with the baseline, then hand over to `verify-nc-program`.

Rules

- Do not hard-code framework or SDK versions, and do not replace template package references with guessed ones.
- Do not edit before the unchanged template builds.
- Keep your run configuration separate from the project sources: no agent wiring in the `.csproj`, no generated NC inside the source tree.
- Do not treat a zero exit code as verification, and do not report success without an NC file you compared.
- Do not change machine settings and do not transmit an NC program.

Done means

The template copy builds unchanged with its SDK reference preserved, the typed change builds, a run through DotNet Posts or the batch runner produced an inspectable report and an NC file (or the limitation is recorded), the output was compared with the baseline, and the remaining verification is stated as still required.

Develop SPPX postprocessors

When to use

Changes to an SPPX postprocessor: a command handler, a subroutine, an object, registers, masks, modal output, separators or subprogram behaviour. Run `inspect-cldata` first and `verify-nc-program` afterwards.

Language foundation

Load `sppx-language-foundations` before interpreting or planning SPPX code. It provides compact language foundations for state, objects, runtime data, registers and output; it does not replace actual CLData, current postprocessor code, or versioned documentation.

Use the foundation for ordinary language forms. If the foundation, current code and actual CLData do not prove a required SPPX/InP semantic, follow the change-critical-unknowns workflow below: read the relevant documentation page before planning or editing.

Tools and the order they go in

The InP MCP server (`inp-mcp-server.exe`) drives the postprocessor IDE. `pp_ping` first, always — it reports the server, the known instances and the default target.

Call	Purpose
`pp_ping`	Health and instance list; call before anything else
`pp_launch`	Start an instance: `mode="windowed"` (default) or `"headless"`; `post_path` opens a postprocessor at once
`pp_instances` / `pp_select_instance`	List instances and pick a default; every call also takes `pid`
`pp_open_post` / `pp_create_post`	Open or create a postprocessor in the connected instance
`pp_get_structure`	Handlers, subroutines, objects, channels — item names come from here
`pp_get_code` / `pp_set_code`	Read an item; replace its full body
`pp_get_registers` / `pp_set_registers`	Register definitions

Call	Purpose
`pp_translate`	Compile. Run after every edit and fix every reported error before continuing
`pp_open_cld` + `pp_run`	Load a CLData project, then interpret it; `pp_run` returns the NC output
`pp_close` / `pp_kill` / `pp_delete`	Close an instance; force it down; delete an item — the last two are destructive

Item kinds: `0` handler, `1` subroutine, `2` object. Handlers exist one per CLData command: they can be edited but not created or deleted. Only subroutines and objects can be created (`create=true`) or deleted.

If no instance is running, prefer `pp_launch` windowed when a person is watching, headless for batch work. If the server reports that the instance manager is unavailable, ask the user to open InP from the CAM system. If several instances are running, pass `pid` explicitly — an ambiguous call fails by design. Calls to one instance are serialized; `pp_translate` and `pp_run` can take minutes.

Workflow

1. `pp_ping`. Get an instance, then `pp_open_post` if the postprocessor is not already open.
2. `pp_get_structure`. Find the handler for the CLData command from `inspect-cldata`. Remember that output may be emitted further down — in a subroutine, a mask or on the next modal change.
3. Read before writing. `pp_get_code` for every item you will change, `pp_get_registers` if registers are involved. Explain the current path from command to NC block before proposing an edit.
4. Resolve change-critical unknowns. Before planning an edit, make a short internal list of every fact the edit depends on but that is not yet evidenced: unclear request terms, unexplained code paths or side effects, unfamiliar SPPX constructs, CLData-to-output mapping, register/mask/format semantics, modal and reset lifecycle, version-sensitive InP behaviour, and conflicts between project data, code, notes or donor postprocessors. For each item, keep the unresolved fact, why it matters, required evidence, fact found and source, and status: `resolved`, `not applicable`, or `blocked`. Use actual CLData for input values, current code and a baseline run for current behaviour, authoritative documentation for SPPX/InP semantics, workspace notes for local conventions, and donor code only as a candidate example. Resolve every change-critical item before planning. For an unfamiliar SPPX construct, built-in object, method, syntax form or InP semantic, an item becomes `resolved` only after `cam_read_doc` has returned a documentation page and the agent has obtained a specific applicable fact from it. Use semantic documentation search for an unfamiliar concept and keyword search for an exact known term, then read the returned page in full. A search snippet, an intended lookup, general programming knowledge, the task description or a similar donor fragment does not resolve the item. Do not produce an implementation plan that names or relies on such a construct before this read succeeds. If the required documentation cannot be read, mark the item `blocked` and report

the concrete limitation rather than proposing syntax. Do not plan or make an edit that depends on a `blocked` item. Keep this list internal unless it blocks safe completion.

5. Generate the baseline if there is none: `pp_open_cld` then `pp_run`, and keep the NC output separately from any reference NC program. Record the InP version, instance or `pid`, postprocessor path, CLData project path, baseline output location, and translation/run messages.
6. Plan the smallest edit and state the expected NC blocks. Cover omitted, repeated and boundary values plus every applicable state transition: first use, modal repeat/change, operation or tool boundary, subprogram boundary and reset.
7. `pp_set_code` with the full body — the code you read, with your change applied. It overwrites; a fragment destroys the rest of the item. Preserve register order and formatting, modal behaviour, separators, the state kept in `Common` and in local variables, and the existing subprogram conventions. Before borrowing code, verify its CLData input, register/mask meanings, modal-state assumptions and target control-system output against the current project.
8. `pp_translate`. Fix every error. Report errors in postprocessor terms — frame, register, handler, G/M code — not as software stack traces. If `pp_set_code` or `pp_translate` fails, reread the saved item before the next change and compare it with the body read before editing; do not assume a failed call left it unchanged.
9. `pp_open_cld` + `pp_run`. Record the changed output location and messages. Compare the new NC output with the baseline and with the reference program, and trace every difference, including unintended ones, to a command and a handler. Maintain the compact internal chain `CLData command and values -> changed SPPX item -> expected NC block -> observed NC difference -> evidence location`; reopen an assumption when generated NC contradicts its expected result.
10. Show the user what changed and hand over to `verify-nc-program`.

Showing your work in VS Code

```
vscode://postprocessor-tools.sppx-tools/goto?post=<path>&handler=
<name>&line=<n>&beside=true
vscode://postprocessor-tools.sppx-tools/highlight?handler=
<name>&line=3&endLine=7&message=<text>
vscode://postprocessor-tools.sppx-tools/run?post=
<path>&config=Agent%20check&create=true&cldata=<path>&ncFilePath=<path>
```

Run links with `code --open-url "<link>"` and URL-encode the values. Address code by `handler` plus `line` or `find`, not by a raw file line. Highlights are cleared by any edit, so set them after the edit, not before. Inside VS Code the same actions are available as the `sppx.goto`, `sppx.highlight`, `sppx.symbols`, `sppx.whereAmI`, `sppx.status` and `sppx.run` commands, which return their result instead of just opening a view.

`pp_run` is your own silent check; the Generate NC panel is what the user watches. Create your own run configuration (`create=true`) rather than modifying the user's, and read `sppx.status` first to learn the post input parameter keys before setting values.

Rules

- Never edit an item you have not read in this session.
- Never claim completion without `pp_run` output that you compared.
- Do not rewrite a whole postprocessor for a local fix, and do not silence a compile error by deleting code.
- `pp_delete` and `pp_kill` are irreversible: ask the user first, and prefer `pp_close`.
- Do not put .NET property syntax into SPPX code, and do not carry register defaults or CLD mappings over from another postprocessor unchecked.

Done means

Read-before-write is evidenced, the diff is minimal, compilation is clean, `pp_run` produced NC output on the recorded project, the output was compared with the baseline and the reference, every difference is explained, and the remaining verification is stated as still required.

Inspect CLData

When to use

Whenever the task depends on the input data: a CLData command, a section, a parameter, the machine, the units, or the reason a block appears in the NC program. Always before editing an SPPX handler or a .NET handler.

Tools

The CLData MCP server (`InpCoreMCP.exe`) is read-only and works without a running CAM system. The calls used below:

Call	Purpose
`cld_open_project`	Open a `*.stcp` / `*.stc` project or a `*.inpcld` file; returns the project id, the machine, the units and the file list
`cld_list_files`	Files of the project with their command counts
`cld_get_skeleton`	Section tree of one file plus a histogram of command types inside each section
`cld_list_commands`	A range of commands as a listing, with `include_names` / `exclude_names` filters
`cld_get_command`	One command in full: raw array, named parameters, typed projection
`cld_find_command`	Find a command by name, optionally filtered by values
`cld_get_unique_command_names`	Which commands the project contains at all — that is, which handlers the postprocessor needs
`cld_find_parameter` / `cld_get_parameter`	Locate a parameter by name, then read it by exact path
`cld_get_machine_info` / `cld_get_project_parameter`	Machine description and project-level settings

If no CLData tools are available, say so and ask for a reduced export; do not reconstruct the data from documentation.

When the answer is not in the CLData at all — how the machine is configured, what an operation was set to, which tool is mounted — and CAM tools are available, read it from the source project instead of inferring it. The same tools can prepare the input when the fixture you need does not exist yet: recalculate the toolpaths, export the CLData of the active project, create or open a test project. Say when you did this: regenerated data invalidates every command index you reported earlier.

Workflow

1. `cld_open_project` on the project you were given. Record the project id, machine, units and file list, and report them — a mismatch here means you are inspecting the wrong data.

2. `cld_list_files` and check `command_count`. Never start by dumping a large file.

3. `cld_get_skeleton` for the file of interest. The skeleton shows the section tree and, in NC-subroutine files, the subroutine boundaries, with a per-section histogram. Note that it is order-free: read a range for order. On files with hundreds of thousands of commands this call is heavy but worth it — it also makes later indented listings cheap.

4. `cld_list_commands` over a bounded range to see order and context. Exclude the high-volume motion commands when they are not the subject (`exclude_names: ["GOTO"]`), or include only what you are looking for.

5. `cld_get_command` for the command itself. For a large parameter tree, use `cld_find_parameter` to locate the path and `cld_get_parameter` to read it, extending the path level by level instead of dumping the whole tree.

6. Choose the path dialect for the code you are going to write: `sppx` for the postprocessor language, `dotnet` for the .NET SDK. The same value has different index bases in the two dialects — 1-based in the SPPX form, 0-based in the typed .NET form. Returned paths are paste-ready in the dialect you asked for; do not translate them by hand.

7. Record what you found: file, section, command index, parameter paths, actual values, units, and what is absent. Absence in one fixture is not proof that a command never carries the parameter.

8. Show it to the user when it matters. The CLData Inspector opens the exact command:

   ```
   vscode://postprocessor-tools.cldata-inspector/reveal?project=<path>&file=
   <index|name>&command=<index>&parameter=<name>&where=both
   ```

   Run it with `code --open-url "<link>"` and URL-encode the values. Inside VS Code the `cldata.reveal` command does the same and returns what was selected.

Rules

- No full-file scans without a skeleton and a bounded range.
- Do not guess a subtype, an array index, a parameter name or a collection position — read it.
- Command indexes belong to one revision of the data. After the project is regenerated the server rereads it and asks you to repeat the call; results captured earlier are stale. Always report indexes together with the project.
- Never modify the project, its settings or the CLData while inspecting.
- Do not disclose unrelated customer data; ask for a reduced fixture instead.

Done means

The project, file, section and command are identified; the required values were read through verified calls and reported with their paths and dialect; units and machine are known; and everything still unknown is listed as unknown, not filled in from documentation.

Postprocessor development

When to use

The user wants to create, adapt, debug or review a postprocessor, or asks why an NC program looks the way it does. Start here, then hand over to `inspect-cldata`, `develop-sppx-postprocessor`, `develop-dotnet-postprocessor` and `verify-nc-program`.

Establish the facts first

Before planning anything, know these five things. Ask for what is missing instead of assuming it.

1. Postprocessor type — SPPX (`*.sppx`, `*.spp`, `*.inp`) or .NET (a C# project or a compiled assembly). The tools and the whole workflow differ.
2. The postprocessor — the file or project path, and whether it is a copy you may change.
3. A test project — a CAM project (`*.stcp`, `*.stc`) or a `*.inpcld` file with a short, representative toolpath.
4. The requested change, expressed in NC output: which block is wrong, what it should be, on which machine and control system.
5. A reference — the current NC program, and a known-good program if one exists.

Then check which tools you actually have: CLData tools (`cld_*`), InP tools (`pp_*`), CAM tools if the CAM server is connected, the VS Code extension commands, a `dotnet` build tool, documentation as local Markdown or through a retrieval service. Report what is missing rather than substituting a guess for it.

CAM tools are secondary here, but they answer what CLData cannot: the machine configuration, an operation's parameters, the tool that is mounted. They can also build the input — recalculate the toolpaths, export CLData, create a test project — when the fixture you need does not exist yet.

Trust order for facts

1. Values read from the actual project and results of actual runs.
2. The official documentation for the installed release, and the templates shipped with it — authoritative for what the product does.
3. The user's own written practices and decisions in the workspace — authoritative for the conventions of their machines and their team, which no published documentation covers. Where the two seem to disagree, the documentation describes the product and the notes describe this shop; say which one you followed.
4. Distributed postprocessors supplied with the product.
5. Patterns seen repeatedly across an internal postprocessor corpus — evidence, not specification.
6. General knowledge, only when marked as an assumption.

The published documentation, whether searched through the knowledge server or read from a local checkout, is a snapshot: it does not know this shop. The workspace notes are the living part, and keeping them current is part of your work — see the `prepare-workspace` skill.

A CLData command name never determines its parameter values. Read the project.

Workflow

1. Restate the task as an observable difference in the NC program, and name the acceptance check.
2. Run `inspect-cldata` to find the command and its real parameters in the test project.
3. Locate the code that processes it — an SPPX handler or a typed .NET handler — and explain the current path to the output before changing anything.
4. Plan the smallest change that produces the expected output. State the expected blocks, the modal transitions and the edge cases.
5. Apply the type-specific skill to make the change, compile it and run it.
6. Run `verify-nc-program` on the result.
7. Report: files changed, evidence from the project, build and run results, the NC differences with their causes, verification state, and what remains for the user to check.

Rules

- Do not invent CLData mappings, parameter names, SDK properties, tool signatures or register defaults.
- Do not edit an item without reading its current content first.
- Do not change machine configuration, delete items to make something pass, or transmit an NC program anywhere.
- Do not call the task done without a run: compilation and a zero exit code are not results.
- Do not use a full customer project where a reduced fixture works, and keep credentials and customer data out of the conversation.

Done means

The change is minimal and explainable, it is based on data actually read from the project, the postprocessor compiled and ran, the generated NC program was compared with the baseline and reviewed, the verification state is stated explicitly, and the remaining human checks are named.

Prepare the workspace

When to use

At the start of work with a user whose workspace has no `AGENTS.md` and no notes folder, whenever they ask how to organise their files, and at the end of any task — the distilling step below is part of finishing, not an extra.

Assume the user is a postprocessor developer, not a programmer: they may never have used version control, and nothing here may require them to learn it.

What you are building and why

You have no memory between sessions. The workspace is it. Everything the user explained yesterday is gone unless it is written in a file you will read again. Say this plainly when you propose the layout — not as an apology, as the reason.

```
work\
├── AGENTS.md      the standing brief, read every session - keep it short
├── posts\         the postprocessors
├── projects\      test CAM projects and reference NC programs
├── docs\          a copy of the postprocessing documentation
└── notes\         the user's own practices and solved cases
    └── skills\    their own skills, in the same format as the published ones
```

Adapt the names to what already exists; never reorganise a folder the user is working in without asking first.

Workflow

1. Look before proposing. List what is already there: postprocessors, test projects, reference NC programs, an existing instruction file, whether the folder is already a git repository (`git rev-parse --git-dir`).
2. Propose the layout as a short plan, in the user's terms, and say what each part is for. Create only what they agree to, and move nothing without permission.
3. For .NET postprocessors the editor is opened on the postprocessor folder itself, so what lies above it falls out of view: the standing brief is read from the folder you were opened on, not from the ones above it. Put the repository at the top of the whole workspace rather than inside one postprocessor. Then, if the client can be told where the brief is — Kilo Code takes a list of instruction files in the `instructions` key of its global `kilo.jsonc` — add the full path of the main `AGENTS.md` there once. Otherwise leave a two-line `AGENTS.md` in each postprocessor folder pointing at the main brief and at the notes. Install the skills for the user rather than for one project, so they do not depend on which folder is open.

4. Draft `AGENTS.md` from what you can already see and what they tell you: machines and control systems, which postprocessor serves which, conventions that always apply, where test projects and reference programs live, the rules of engagement (read the actual CLData before editing, run and compare before reporting done, never touch a production postprocessor unasked). Keep it to about a page: it is read in full every session. Details belong in `notes\`.
5. Record standing constraints in the user's own words, quoted. A rejected option with the reason for rejecting it is worth more than a description of the chosen one.
6. Give yourself the documentation: connect the knowledge server if it is available, and clone the documentation repository into the workspace as well — the checkout is what pins a revision and works offline. Note the revision you cloned. If a knowledge cache has never been refreshed on this machine, refresh it once.
7. Offer version history — see below — and accept a refusal without arguing.
8. When the task itself is done, distil it: one short file in `notes\`, one topic per file, saying what was wrong, how it was found, what fixed it and why that way. If a way of working repeated, write it down as a skill in `notes\skills\` instead. Then tell the user what you wrote, in one line.
9. If what you wrote applies beyond this shop — a technique, a check, a way of reading the data, with nothing specific to their machines or customers in it — say so, and offer to prepare it for sharing: a self-contained file, cleaned of customer names, part numbers, machine particulars and internal paths, with what you removed reported to the user. Submitting it to the shared knowledge base is done from the CAM Agent chat, so leave the submission to the user unless you are working there yourself. Never publish anything on your own initiative.

Version history

Ask once, plainly: keep history in git, or keep working with folders and copies? Both are legitimate. If the user is uneasy about letting you edit their files, say the useful part rather than reassuring them: under version control nothing you do is irreversible, and any state can be brought back. If they decline, help them the other way — a copy of the postprocessor before a substantial change, reference NC programs kept next to the test projects — and do not raise it again in that session.

If they accept, do all of it yourself:

- check for git (`git --version`); if it is missing, install it — `winget install --id Git.Git` on Windows, or point them at https://git-scm.com/downloads if winget is unavailable;
- `git init`, then a `.gitignore` that excludes generated NC output, build artefacts and temporary files, and a `.gitattributes` that fixes line endings for the text files in the repository;
- verify there are no credentials, licence files or customer archives in the first commit, then commit the current state as the starting point;
- explain in two sentences what they got, without git vocabulary: every state can be brought back, and every change can be explained.

Then serve the plain-language requests without making them learn commands:

They ask	You do
"commit this"	Commit with a message that says why, not what changed
"compare with Friday" / "with the previous version"	Diff the working tree or two revisions, and report differences in NC terms
"when did this break?"	Search the history for the change that introduced the behaviour; bisect with the test project if the history is long
"why is this line here?"	Find the change that added it and quote its message
"put it back as it was"	Restore the file from history — never retype it from memory

Rules

- Never commit passwords, tokens, licence files or customer data. Check before the first commit, not after.
- Never force version control on a user who declined it, and never make a routine task depend on it.
- Keep `AGENTS.md` short. Everything you add to it is paid for in every future session; if it is not needed every time, it belongs in `notes\`.
- One fact in one place, and always the current state. No "it used to be" asides in notes — that is what history is for.
- Do not describe a task as finished while its outcome exists only in the chat.
- Do not restructure or rename the user's existing folders on your own initiative.

Done means

The workspace has a layout the user agreed to, `AGENTS.md` exists and is short and accurate, the documentation is reachable, `notes\` contains at least the outcome of the current task, and history is either under git with a clean first commit or explicitly declined by the user.

SPPX language foundations

Purpose and boundary

Use this skill when reading or planning SPPX code. It supplies stable language concepts and small syntax patterns; it does not replace the development workflow, CLData inspection, InP MCP tools, NC verification, or the documentation MCP.

- `postprocessor-development` frames the task and source trust order.
- `inspect-cldata` reads actual command values and parameter paths.
- `develop-sppx-postprocessor` controls InP work, editing, translating, running and comparison.
- `verify-nc-program` assigns the verification state.
- This skill explains common SPPX forms so that general C# / Pascal / Delphi knowledge is not substituted for the postprocessor language.

For an unfamiliar or version-sensitive SPPX/InP semantic, read the authoritative documentation page before relying on it. Current post code, actual CLData and a baseline run remain stronger evidence for the current project than a generic pattern here.

Execution model

An SPPX postprocessor transforms a stream of CLData commands into NC blocks:

```
CLData command -> enabled handler -> state and registers -> OUTBLOCK / FORMBLOCK /
MASK -> NC block
```

- A CLData token and a handler name are not necessarily identical. For example, the token may be `GOTO.abs` while the current post's handler is `AbsMov`. Obtain handler names from `pp_get_structure`; never construct one from an NC word or a CLData token.
- A handler can set state without printing a block. Output may occur in a called subroutine, a mask, `Filter`, or a later modal transition.
- Register assignment alone does not prove that an NC word is emitted. The output operation and modal state determine the resulting block.
- `COMMON` executes before normal command processing. `PARTNO` is commonly used for program-start initialization, but its actual role must be read in the current post.

State: choose lifetime before code

Use the lifetime required by the behavior, not the most convenient syntax.

Need	Common location	Verify before use
Temporary value used only during one handler invocation	Local variable in that handler or subroutine	The value is not needed by another handler or later command
State shared by handlers or preserved across commands	`Common`	Initialization and reset boundaries in the current post
Structured state with related data and behavior	Custom object instance	Object syntax and instance lifetime; documentation if unfamiliar
An NC address to be formatted/output	Register list	Existing binding, list position, format and modal behavior

Do not put an NC register in `Common` merely because its value must persist. Do not create a register merely because a local calculation needs storage. Variables ending in `@` conventionally represent previous register state; treat their semantics as runtime behavior and do not redeclare them without documentation/current-code evidence.

Common lifecycle questions

Before adding or resetting state, identify which transition owns it: program start, operation boundary, tool change, first motion, subprogram boundary, command repeat, or program end. Common patterns such as `PARTNO`, `PPFUN`, `LoadTl`, `Fedrat`, `AbsMov` and `Filter` are only routing clues. The real CLData and current post decide where a reset belongs.

Small verified syntax patterns

These patterns demonstrate language form only. They do not prove that the shown storage, reset point, handler or output behavior is suitable for a particular post.

Variables, conditions and blocks

```
Count: Integer;
FeedValue: Real;

if Count > 0 then begin
  FeedValue = FeedValue + 1
end
```

- Purpose: declaration and a multi-statement conditional block.
- Guaranteed: SPPX uses typed declarations and `begin ... end` for grouped statements.

- Not guaranteed: the correct type, variable scope, condition or lifecycle for the current post.

Arrays and records

```
Values: array 3 of Real;
State: record
  Ready: Integer;
  LastValue: Real
end;

Values[1] = 10;
State.Ready = 1
```

- Purpose: basic collection and structured data form.
- Guaranteed: array and CLD-style indexing are 1-based in SPPX.
- Not guaranteed: an index exists in actual CLData, an array size is sufficient, or a record is better than `Common`/an object.

Custom object form

```
object FeedState
  LastFeed: Real;
  Initialized: Integer;

  procedure Reset
  begin
    self.LastFeed = 0;
    self.Initialized = 0
  end
end.

State: FeedState;
State.Reset
```

- Purpose: fields, a method, `self`, object closing syntax and an instance declaration.
- Guaranteed: custom object form shown here is a language pattern.
- Not guaranteed: whether an object should be used, where its instance belongs, when `Reset` is called, or how feed/modal output works. Read documentation before first use of an unfamiliar object form or method semantic.

Subroutine form

```
sub PrepareOutput(Value: Real)
  if Value <> 0 then begin
    PRINT STR(Value)
  end
subend
```

```
call PrepareOutput(10)
```

- Purpose: subroutine definition and call form.
- Guaranteed: a `sub ... subend` body can encapsulate repeated SPPX logic.
- Not guaranteed: that `PRINT`, output, parameters or the chosen ownership are appropriate for a production handler.

Runtime input data

Runtime item	Use	Rule
`CLD`	Numeric parameters of the current command	Values and indices come from `inspect-cldata` and current handler/docs evidence
`RecNum`	Count of current numeric CLD parameters	Check before reading optional positions
`CLData$`	Text data for textual commands	Do not treat it as a numeric CLD value
`Cmd`	Named, textual, typed and complex command parameters	Prefer named access when actual command data provides it
`Project`	Project-level parameters	Read actual project settings; do not infer them from NC
`GMA`	Machine-axis data for relevant multi-axis commands	Use actual machine/schema evidence; do not invent axis names
`RGS`	Register access by index	Read the current register model before relying on indexed access

Typical access forms:

```
X = CLD[1];
Name$ = Cmd.Str["Comment"];
Changed = Cmd.Int["ToolChanged"];
Value = Cmd.Flt["SomeValue"];
```

`CLD[1]` in this example is only syntax. It never proves that position 1 means X, feed, a mode, or any other value for the current command. `inspect-cldata` returns SPPX-ready 1-based paths; do not convert

them manually from .NET paths.

Registers and NC output

A register combines an NC identifier, an internal name, a current value, previous state and output format. `OUTBLOCK` normally scans the register list in list order and emits changed values in that order.

```
register list order + changed current/previous state -> NC word order in OUTBLOCK
```

Consequences:

- The order of assignments in a handler does not define NC word order.
- Different registers may share an NC identifier when the post needs separate output positions; choose by the current register list and intended sequence, not the first matching identifier.
- Before adding a register, read `pp_get_registers`, locate current bindings and verify that no existing register covers the required address and position.
- Do not reorder existing registers to solve one local block. A new register, if truly needed, must have a validated position and format.
- Formatting, scale, signs, leading/trailing zero policy and duplicated-register semantics are post/control specific. Read the current definition and documentation before changing them.
- `X@`, `F@` and similar previous-state variables are not ordinary independent registers. Do not reset or declare them based only on generic programming intuition.

Output operations

Operation	Practical meaning	Must verify
`OUTBLOCK`	Forms and emits a normal NC block from current register/output state	Which registers are eligible, list order and modal state
`FORMBLOCK`	Forms an output block without normal NC emission	Destination and later use of the formed string
`MASK(...)`	Produces output according to mask-specific syntax and rules	Mask semantics, modifiers and state effects from docs/current code
`OUTPUT`	Writes a direct text block	File/destination and control-specific formatting

Operation	Practical meaning	Must verify
`Filter`	May transform a block after formation and before final NC insertion	Current signature and existing post behavior

Do not put mask-only modifiers into ordinary handler expressions. A mask, its register resolution and modal effects are not inferred from a similar fragment; read its documentation/current definition first.

CLData and handler routing clues

These are navigation clues, not universal implementation recipes.

Intent area	Often related handlers/data	Do not assume
Linear motion	`AbsMov`, `MultiGoto`, `PhysicGoto`, `From`, `GoHome`	Which coordinates/axes apply or where output occurs
Feed	`Fedrat`, then a later motion handler	Whether feed emits immediately or is modal
Spindle	`Spindl`	Control codes, orientation support and output placement
Tool change	`LoadTl`, command data such as `ToolChanged`	Reset boundaries or T/M code sequence
Program/operation/subprogram lifecycle	`PartNo`, `PPFUN`, `Structure`, `FINI`	Exact CLD subcodes and local conventions
Cycles	`Cycle`, `ExtCycle`	Field mapping and first/repeated-cycle output
Coordinates/interpolation	`Origin`, `Interpolation`, `Circle`	Plane, machine transform and kinematic assumptions

For exact CLD mappings, use `inspect-cldata` first. Read detailed documentation for unfamiliar command variants, cycles and optional fields.

Built-ins and documentation trigger

The following categories are common, but their methods and version behavior are not memorized by this skill:

- string/date helpers;
- `Point3D`, `Matrix`, `EulerConverter`, `Quaternion4d` and transformation helpers;
- `NCChannel` and multichannel state;
- `Cmd.Ptr` complex values and machine-axis structures;
- masks, formats, register modifiers and output filters.

Before first use of a built-in object, method, unfamiliar runtime object, mask form or version-sensitive feature:

```
cam_search_docs_semantic or cam_search_docs
-> cam_read_doc
-> use the specific documented fact
```

A search snippet, a skill example or similar code from another postprocessor is not the language specification. If the page cannot be read and the semantic is necessary, leave it `blocked` rather than inventing syntax.

Common mistakes

- Writing C# / .NET property syntax into SPPX.
- Constructing a handler name from a CLData token or NC word.
- Guessing `CLD[...]` indices, optional fields or `Cmd` parameter names.
- Treating an assignment as proof of output without tracing `OUTBLOCK`, masks and modal state.
- Reusing a donor's global state, register name, format, reset point or output fragment without checking the current post.
- Treating a successful translation as proof of correct NC behavior.
- Using a language mini-pattern as a production handler template.

Use with the workflow

Load this skill to recognize ordinary SPPX forms while reading code. Then follow `develop-sppx-postprocessor` for the actual change. If this skill, the current post and actual CLData do not prove the semantic needed by the change, the workflow's change-critical-unknowns gate requires documentation lookup before planning or editing.

Verify NC programs

When to use

Before any generated or changed NC program is presented as a result — SPPX or .NET, regardless of how cleanly it built.

What you need

The source diff, the postprocessor and tool versions, the project the program was generated from, the baseline NC program, the new NC program, the machine and control system, and the expected behaviour. If the baseline is missing, generate one from the unchanged postprocessor before comparing.

State how far the check went

Report exactly one of these, and never upgrade it without the evidence:

State	Requires
Not verified	Only a plan, a diff or a build exists
Technically verified	The run completed, and the log, result code and generated file are kept; the NC content has not been reviewed
Verified on a test project	Baseline and new program compared on the test and boundary sections, machine parameters checked
Approved for production	A qualified person confirmed it for a specific machine and control system — you cannot assign this state

Workflow

1. Check that the diff contains the requested change and nothing else, and that output file name, encoding, line endings and error handling are as expected.
2. Reproduce the run from the recorded project. Exercise the cases the fixture does not cover as far as the data allows: omitted, repeated, maximum and unsupported values.
3. Diff the programs and start at the first differing block. Trace each difference to a CLData command and the code that produced it. Unintended differences matter more than intended ones.
4. Review the program as a whole: machine, control system, units and work coordinate system; program start and end; tool changes, spindle, feeds, coolant, safe positions; signs, axis order,

compensation, work offsets, retracts; cycles, subprograms, rotary motion, limits, synchronization; and whether the program still corresponds to the original toolpath.
5. Name what only a person or a simulator can settle, and say so plainly instead of implying it was covered.
6. Report the state from the table, with the evidence and its location.

Rules

- Compilation, a zero exit code, a plausible-looking listing and a matching text diff are not verification.
- Do not suppress warnings to make output pass, and do not change machine configuration or postprocessor settings to make a comparison succeed.
- Do not transmit a program to a control system, and do not describe a result as safe or production-ready.
- Do not claim simulation or machine checks that were not run.

Done means

The source and NC diffs were reviewed, differences are explained at command level, the reachable edge cases were exercised, the verification state is stated with its evidence, and the checks left to the user — simulation, machine verification, approval — are named explicitly.