APL AutoSoftware Programming Language

(C) 2001-2008 by AutoSoftware srl
08/22

language reference

for post-processors, macros, rebuild processor, barcode processor, pre-processors

This document describes commands and variables used as internal language to create post-processors, macros and other internal available interfaces.

To create **processors** use commands and variables described below.

To create a **macro**, use the following basic / expanded / dialog commands and the macros specific commands described in the macro commands and variable section.

Basic commands

NOTE

Variables must be included in square brackets:	[A]
Variables names are case insensitive:	[Xc] = [xc] = [xC] = [XC]
Data types: simple variables, structures, arrays	
~LET	
~dimAry	
~delAry	
~Ary	
~LETARY	
~RENARY	
~COPYARY	

command: ~IF

conditional statement

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOR, ~DO, ~, ~I

syntax

```
~IF <condition>
  <condition verified code>
~ELSE
  <condition not verified code>
~ENDIF
```

note

<condition> is a string evaluation, only the **=** and **!=** operators are supported; use this command in conjunction with ~() and ~I()

example

```
~IF ~([d] > 0 AND [d] < 2 * [r]) = 1
    ~LET [ang] = ~(acos([d]/(2*[r])))
~ELSE
    ~LET [ang] = 60
~ENDIF
```

command: ~FOR ... ~NEXT

Loop statement

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~DO, ~IF, ~, ~I

syntax

```
~FOR <variable> = <from> TO <end>[ STEP <step>]
```

```
    <loop body>
    ~EXIT_FOR
    <loop body>
  ~NEXT
```

note

* The ~FOR execution algorithm is as follows:
  1) <variable> is assigned the <from> value.
  2) <variable> is checked if the <end> has been reached.
  3) the next instruction is then executed or, if the <end> has been reached, the ~FOR/~NEXT loop is skipped.
  4) go to step 2)
* When the ~NEXT instruction is encountered, <variable> is incremented by the <step> value and then evaluated if the <end> has been reached.
* STEP is optional, if not specified <step> = 1.
* <from>, <end> and <step> can be expressions. They are automatically integer evaluated. You do not need to set them as ~I(...).
* <from> and <step> are evaluated at the beginning of the ~FOR, <end> is evaluated at the moment <variable> is evaluated.
* After the ~NEXT, <variable> has a value that depends upon <step>, <end> and any eventual ~EXIT_FOR statement.

example1

```
 ~FOR [idx] = 0 TO [qt] - 1
   ~LET [A] = ~I([A]+[idx])
 ~NEXT
 #now [idx] = [qt]
```

example2

```
 ~LET [idx] = 0
 ~LET [A] = 0
 ~FOR [idx] = [idx] TO [A] >= [idx] STEP 1
   ~LET [A] = ~I([A]+[idx]-1)
 ~NEXT
```

command: ~DO ... ~LOOP

Loop statement

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOR, ~IF, ~, ~I

syntax1

```
  ~DO WHILE(<condition>)
    <loop body>
    ~EXIT_LOOP
    <loop body>
  ~LOOP
```

syntax2

```
  ~DO
    <loop body>
    ~EXIT_LOOP
    <loop body>
  ~LOOP UNTIL(<condition>)
```

note

<condition> is a string evaluation, only the = and != operators are supported, use this command in conjunction with ~() and ~I()

example1

```
 ~LET [idx] = 1
 ~DO
   ~LET [idx] = ~([idx] - 1)
 ~LOOP UNTIL(~([idx] != -1) = 1)
```

example2

```
 ~DO WHILE(~([ang] < 0) = 1)
   ~LET [ang] = ~([ang] + 360)
 ~LOOP
```

command: ~CALL

Call a subroutine

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~START, ~GOTO, ~EXECUTE

syntax

```
  ~CALL <sub name>
```

note

The <sub name> can be a variable.

example1:

```
 ~CALL MY_G0
```

example2:

```
 ~LET [SUB_TO_BE_CALLED] = MY_G0
 ~CALL [SUB_TO_BE_CALLED]
```

command: ~START

Subroutine definition

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CALL, ~?, ~EXIT_SUB

syntax

```
~START <sub name> [~MODAL(<string1>,...)]
  <sub body>
  ~EXIT_SUB
  <sub body>
~END
```

note

The optional ~MODAL statement is used for removing unchanged variables from the ~? command.

example

```
~START HOLE ~MODAL(X=,Y=,Z=,E=,V)
  ~?B X=[X] Y=[Y] Z=[Z] V[FEED] T[TOOL]
~END
```

command: ~EXIT_SUB

Subroutine exit

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~START

syntax

```
~START <sub name> [~MODAL(<string1>,...)]
  <sub body>
  ~EXIT_SUB
  <sub body>
~END
```

example

```
~START HOLE
  ~IF [ret] == 0
    ~EXIT_SUB
  ~ENDIF
  ~?B X=[X] Y=[Y] Z=[Z] V[FEED] T[TOOL]
~END
```

command:

Remark

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

```
# <Remark text>
```

note All characters after the # are ignored.

example

```
  # This is a comment line
```

command: ~LET

Variable assignment

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~(), ~I(), ~STRCAT()

syntax

```
~LET <variable name> = <expression>
```

note

<expression> is not evaluated. Use ~(), ~I() or ~STRCAT() for numeric and string expression evaluation.

example

```
  ~LET [y] = 1
```

command: **~GOTO**
Unconditional jump

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~:, ~CALL, ~IF

syntax
~GOTO <label>

example
```
~GOTO Door_Open
~LET [lamp] = 1
~:Door_Open
```

command: **~:**
Label for GOTO

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GOTO

syntax
~:<label>

example
```
~GOTO Door_Open
~LET [lamp] = 1
~:Door_Open
```

command: **~DEBUG**
Program debugging

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~DEBUG <expression>

note
Send output to debug.txt file.
In order to enable the ~DEBUG command, the following procedure must be followed:
From the Program supervisor, select: Utility->Technical Menu->Defaults
add: name: DEBUGPP
value: 1

example
```
~LET [A] = 1
~DEBUG The A value is [A]
```

command: **~MSGBOX**
Send output to a Message Box

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~MSG

syntax
~MSGBOX(<title>, <buttons>, <expression>,...)

input
<buttons> :

0 => OK	1 => OK + Cancel
2 => Abort + Retry + Ignore	3 => Yes + No + Cancel
4 => Yes + No	5 => Retry + Cancel

You can add Windows specific flags:

MB_ICONHAND	16	MB_ICONQUESTION	32
MB_ICONEXCLAMATION	48	MB_ICONASTERISK	64

return

1 if the OK or Yes button was selected	2 if the Cancel button was selected
3 if the No button was selected.	4 if the Abort button was selected

5 if the Ignore button was selected	6 if the Retry button was selected

example
```
~LET [A] = 1
~LET [B] = 1
~MSGBOX("MessageBox test", 0, "A = ", [A], [CHR13], "B = ", [B])
```

command: ~MSG
Retrieve a message

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~MSGBOX, ~EXTRA

syntax
~MSG(<message>)

input
<message> :=
the message number identifier or the message string or a common message identifier (see 'APPENDIX A').
return the message string.

note Useful in multi-language environments.

example
```
~MSG(1)          => gets message at line 1 from the message file
~MSG("my string") => leave the string as is (untraslated)
~MSG(warning_msg) => returns the “Warning” message in the current language
```

command: ~()
Evaluate a Floating-Point expression

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~I(), ~GETV(), ~STRCAT(), ~IF, ~DO ... ~LOOP

syntax
~(<math expression>)

input the expression to be evaluated
return the result

note Can be used in loop and conditional commands. See 'APPENDIX B' for a list of operators.

example
```
~LET [X0] = ~([X0] + [DIAM] * COS([angb]))
```

command: ~I()
Evaluate an Integer and/or boolean expression

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~(), ~GETV(), ~STRCAT(), ~IF, ~DO ... ~LOOP

syntax
~I(<Integer/boolean expression>)

input the expression to be evaluated
return the result

note Can be used in loop and conditional commands. See 'APPENDIX B' for a list of operators.

example
```
~LET [i] = 0
~DO WHILE(~I([i] < 10) = 1)
  ~LET [i] = ~I([i] + 1)
~LOOP
```

command: ~EVAL
makes a math evaluation using the number of specified digits

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GetDecimalPrecision, ~ SetDecimalPrecision

syntax

~LET [result] = ~Eval(<number of decimal digits>, <expression>)

input

<number of decimal digits>	:= the number of decimal digits
<expression>)	:= the expression to be evaluated

return

The number resulting from the evaluation of <expression>. The number will be represented with <number of decimal digits> number of decimal digits.

example

```
~LET [ndec] = ~GetDecimalPrecision()
~SetDecimalPrecision(1)
~LET [x0] = ~(12)
~LET [x1] = ~Eval(5, [x0])
~LET [x2] = ~(12)
~MSGBOX(,,"x0=",[x0]," x1=",[x1]," x2=",[x2])
~SetDecimalPrecision([ndec])
```

command: **~?, ~??**

Send output to file

applied to:	post-processor	macro	drw rebuild
	X		

See also: ~SetOutPutFile, Appendix C

syntax

~? <expression>
~?? <expression>

note

~? => Interprets line and then send to output file
~?? => Send line, as is, to output file

example

```
~?? (This line will be printed as is: [ZS] will not be evaluated)
~? (Xs=[XS], Ys= [YS] and Zs[ZS])
```

command: **~GetDecimalPrecision**

gets the number of decimal digits used for computations

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GetDecimalPrecision, ~Eval

syntax

~LET [deci] = ~GetDecimalPrecision()

example

```
~LET [ndec] = ~GetDecimalPrecision()
~SetDecimalPrecision(1)
~LET [x0] = ~(12)
~LET [x1] = ~Eval(5, [x0])
~LET [x2] = ~(12)
~MSGBOX(,,"x0=",[x0]," x1=",[x1]," x2=",[x2])
~SetDecimalPrecision([ndec])
```

command: **~SetDecimalPrecision**

Changes the number of decimal digits used for computations

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GetDecimalPrecision, ~Eval

syntax

~SetDecimalPrecision(<number of decimal digits>)

input

<number of decimal digits> *: new number of decimal digits for floating point numbers*

example
```
~LET [ndec] = ~GetDecimalPrecision()
~SetDecimalPrecision(1)
~LET [x0] = ~(12)
~LET [x1] = ~Eval(5, [x0])
~LET [x2] = ~(12)
~MSGBOX(,,"x0=",[x0]," x1=",[x1]," x2=",[x2])
~SetDecimalPrecision([ndec])
```

command: ~SetOutPutFile

Changes the output file

applied to:	post-processor	macro	drw rebuild
	X		

See also: ~?

syntax

~SetOutPutFile(<New OutPut File>[, <Open type>])
~SetOutPutFile()

input

<New OutPut File> : new file that will be used for output
<Open type>: "wt" (write) or "at" (append); if not specified, "wt" is assumed

return *the Output file name with full path*

note

The output file is set in the C++ code, but can be modified by this function.
~SetOutPutFile() without parameters restores the default C++ Output file.

example
```
~SetOutPutFile()
~? ;Creating file: INI.PRM
~SetOutPutFile(INI.PRM)
```

command: ~FOPEN

Open a file for reading/writing

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPENCOMM, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax

~LET [fh] = ~FOPEN(<FileName>, <OpenMode>, <NetMode>, <CreationMode>)
~LET [fh] = ~_FOPEN(<FileName>, <OpenMode>, <NetMode>, <CreationMode>)

input

<FileName> := File name with optional Full Path
Valid names are:
~FOPEN("C:\WINDOWS\Test.txt", “WR”, “SNO”, “OA”)
~FOPEN("\\.\LPT1:", “WR”, “SNO”, “OE”)

<OpenMode>

“RD” = Open file for reading	“WR” = Open file for writing
“RW“ = Open file for reading and writing	

<NetMode>

“SRD” = Allow Read to others	“SWR” = Allow Write to others
“SNO” = Lock file (no sharing)	

<CreationMode>

“CN” (Create New)	“CA” (Create Always)
“OE” (Open Existing)	“OA” (Open Always)
“TE” (Truncate Existing)	

return the file handle. -1 if error: see note for ~_FOPEN().

note

See the Windows SDK CreateFile() documentation for further details.
~_FOPEN() means that no error is given by the decoder in case the ~_FOPEN() fails. The user must check

the returned value:
-1 means 'Device not opened', any different value is Ok.

example
```
~LET [fh] = ~FOPEN("Test.txt", “WR”, “SNO”, “OA”)
~FWRITE([fh], "Hello world !")
~FCLOSE([fh])
```

command: ~FDIALOG, ~_FDIALOG

open the default File Open Form or File Save As form

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPENCOMM, ~FOPEN, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax
~LET [fname] = ~FDIALOG(<dlgtype>, <dlgtitle>, <extension>, <extension_msg>,<file_name>, <flags>, <retvalue>)

input
<dlgtype>

1 - to show a "File Open" dialog box	0 - to show a "File Save As" dialog box

<dlgtitle> : = title for the dialog box
<extension> := this parameter has 2 syntaxes: "simple" and "multiple".
Simple is if you want to handle just one extension, Multiple if you want to handle more than one extension.
"Simple" syntax (default):
- type the default filename extension: what's on the right of the dot. Default: "*".
"Multiple" syntax (you must pass an array):
- fill-in an array with all the extensions, one for each element. Look at the example.
Important note: <extension> and <extension_msg> must use the same syntax: both "simple" or both "multiple".
<extension_msg> := This parameter has 2 syntaxes: "simple" and "multiple".
Simple is if you want to handle just one extension, Multiple if you want to handle more than one extension.
"Simple" syntax (default):
- fill-in an array with all the extension descriptions,
"Multiple" syntax (you must pass an array):
- Put a description in each element, one for each element. Look at the example.
Important note: <extension> and <extension_msg> must use the same syntax: both "simple" or both "multiple".
<file_name> := the initial filename that appears in the filename edit box
<flags> := Common dialog Flags, default:
OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT | OFN_NOCHANGEDIR
<retvalue>

0 return the selected file name with full path	1 return the file path
2 return the file name without path	

return
OK: 0 and <fname> has the selected file; Cancel: 0 and <fname> is empty; a different value is error
note
In case of error the ~_FDIALOG() can be used for debugging purposes;
in this case the returned value is the windows error code returned by the Windows SDK CommDlgExtendedError() function.

example1:
```
~FDIALOG(1, "Open File", "INI", "configuration file",
".\MYFILE.INI")
```

example2:
```
~dimAry("exts", 0, 2)
~dimAry("extd", 0, 2)
~LETARY ("extd", 0) = INI files (*.INI)
~LETARY ("exts", 0) = INI
~LETARY ("extd", 1) = All files (*.*)
~LETARY ("exts", 1) = *
~FDIALOG(0, "Save As File", "exts()", "extd()",
".\MYFILE.INI")
```

command: ~FolderSel

displays a dialog box enabling to select a Shell folder

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FDIALOG

syntax

~LET [ret] = ~FolderSel(<dlgtitle>, "folder_name", <flags>)

input

<dlgtitle>	:= title for the dialog box
"folder_name"	:= variable name that will be filled-in with the result
<flags>	:= BROWSEINFO Flags, default: BIF_USENEWUI (see notes). The list of possible values can be found in the shlobj.h file located in the Windows SDK include directory.

return

1: the folder has been selected	0: the user cancels the operation or in case of fail (see notes)

note

Depending on the Operating system, some flags could not work or cause the function to fail. If the user specifies an impossible folder the function fails.

example

```
~LET [rt] = ~FolderSel("This is the title ", "mydir")
~IF [rt] = 1
  ~MSGBOX(,,"The selected dir is: ",[mydir])
~ELSE
  ~MSGBOX(,,"Selection Cancelled")
~ENDIF
```

command: ~FOPENCOMM

Open a communication port for reading/writing

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FWRITE

syntax

~LET [fh] = ~FOPENCOMM(<Comport>, <HandShake>)

input

<Comport> := a string containing the comm port name, baud rate, data bits, parity, stop bits.
Parity can be one of: no, odd, even, mark, space.
See example.

<HandShake> :=

"No" no handshake	"Xon/Xoff"
"Cts/Rts"	"Dsr/Dtr"

If this parameter is not specified, the default Port handshake will be used.

return The file handle. -1 if error: see note for ~_FOPENCOMM().

note

See the Windows SDK CreateFile() and SetCommState() documentation for further details.
~_FOPENCOMM() means that no error is given by the decoder in case the ~_FOPENCOMM() fails. The user must check the returned value: -1 means 'Device not opened', any different value is Ok.

example

```
~LET [fh] = ~FOPENCOMM("\\.\COM1:9600,8,n,1", "No")
~FWRITE([fh], "Hello world !")
~FCLOSE([fh])
```

command: ~FCLOSE

close a file handle

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FOPENCOMM, ~READB, ~READLINE, ~WRITEB

syntax

~FCLOSE(<fh>)

input

<fh> := File handle returned by the FileOpen function

example

```
~LET [fh] = ~FOPEN("Test.txt", "WR", "SNO", "OA")
~FWRITE([fh], "Hello world !")
~FCLOSE([fh])
```

command: ~FREADB

read from a file in binary mode

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADLINE, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax

~LET [readin] = ~FREADB(<fh>, <qtbyte>[, <format>[, <conversion>]])

input

<fh> := File handle returned by the FileOpen function
<qtbyte> := the numberof bytes to be read
<format> see ~WRITEB
<conversion> all the ~WRITEB conversion operators except STR0T.

return the data that has been reading.

example

```
~LET [fh] = ~FOPEN("Test.txt", RD, SNO, OA)
~LET [str] = ~FREADB([fh], 100, "", "")
~FCLOSE([fh])
```

command: ~FREADLINE

read a text line from a file

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADB, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax

~LET [readin] = ~FREADLINE(<fh>)

input

<fh> := File handle returned by the FileOpen function

return The text line that has been reading.

example

```
~LET [fh] = ~FOPEN("Test.txt", RD, SNO, OA)
~LET [str] = ~FREADB([fh], 100, "", "")
~FCLOSE([fh])
```

command: ~FWRITEB

write on to a file in binary mode

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITE, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax

~FWRITEB(<fh>, <value>, <format>, <conversion>)

input

<fh> := File handle returned by the FileOpen function
<value> := the data that has to be written
<format> see ~WRITEB
<conversion> all the ~WRITEB conversion operators.

example

```
~LET [fh] = ~FOPEN("Test.txt", WR, SNO, OA)
~LET [str] = ~FWRITEB([fh], 101, "", "U8")
~FCLOSE([fh])
```

command: ~FWRITE

write on to a file in text mode

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FTELL, ~FSEEK, ~READB, ~READLINE, ~WRITEB

syntax
~FWRITE(<fh>, <string>)

input
<fh> := File handle returned by the FileOpen function
<string> := the data that has to be written

example
```
~LET [fh] = ~FOPEN("Test.txt", WR, SNO, OA)
~LET [str] = ~FWRITE([fh], "Hello World !")
~FCLOSE([fh])
```

command: ~FSEEK

move the file pointer

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK

syntax
~FSEEK(<fh>, <pos>, <Seek_type>)

input
<fh> := File handle returned by the FileOpen function
<pos> := the position where to move to
<Seek_type> := BEGIN
END
CURRENT

return The current file pointer after the ~FSEEK

example1:
```
~LET [fh] = ~FOPEN("Test.txt", "WR", "SRD", "OA"
~FSEEK([fh], 128);
~LET [tool] = ~FREADB([fh], 1, "", "U8")
~FCLOSE([fh])
```

example2:
```
# open for append
~LET [fh] = ~FOPEN("Test.txt", "WR", "SRD", "OA"
~IF [fh] != -1
  ~FSEEK([fh], 0, "END")
  ~FWRITE([fh], ~STRCAT("Append Test",~CHR(13),~CHR(10)))
  ~FCLOSE([fh])
~ELSE
  ~MSGBOX(,,"Error opening file")
~ENDIF
```

command: ~FTELL

move the file pointer

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FOPEN, ~FCLOSE, ~FREADB, ~FREADLINE, ~FWRITEB, ~FWRITE, ~FTELL, ~FSEEK

syntax
~FTELL(<fh>)

input <fh> := File handle returned by the FileOpen function
return the current file pointer.

example
```
~LET [fh] = ~FOPEN("Test.txt", RD, NA)
~LET [tool] = ~FREADB([fh], 1, "", "U8")
~LET [fp] = ~FTELL([fh]);
~FCLOSE([fh])
```

command: ~CMD

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

command: **~PRINT**

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

Expanded commands

command: **~EXISTV**

check if the variable exists

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~SETV, ~GETV(), ~(), ~I()

syntax

~EXISTV(<varname>)

input The variable to be checked for existence

return

0 if the variable does not exists	1 if the variable exists

example

```
~IF ~EXISTV("A") = 1
  ~LET [return] = [A]
~ELSE
  ~LET [return] = ~CHR(0)
~ENDIF
```

command: **~GETV ~_GETV**

get the variable value

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~SETV, ~EXISTV(), ~(), ~I()

syntax

~GETV(<varname>)
~_GETV(<varname>)

input the variable name whose value has to be retrieved
return the variable name.

example

```
  ~LET [A1] = 10
  ~LET [NAME] = ~STRCAT("A",1)
  ~LET [content] = ~GETV([NAME])
  ~MSGBOX(,,"content = ",[content])
```

command: **~SETV**

set the variable value

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GETV, ~EXISTV(), ~(), ~I()

syntax

~SETV(<varname>)

input the variable name whose value has to be set
return the variable name.

example

```
  ~LET [NAME] = ~STRCAT("A",1)
  ~SETV([NAME]) = 10
```

command: **~WRITEB**

File Write Binary

applied to:	post-processor	macro	drw rebuild

X	X	X

See also: ~READB

syntax

~LET <newoffset> = ~WRITEB(<file>, <offset>, <value>, <format>, <conversion>)

input

<file> file that will be written in Binary mode
<offset> position (in bytes) inside the file (fpos), or
-1 to append to the end of the file.
<value> The value that will be written
<format> II (Intel), MM (Motorola), - (no byte swap)
<conversion>

S8 (integer 8 bit)	U8 (unsigned int. 8 bit)
S16 (int. 16 bit)	U16 (unsigned int. 16 bit)
S32 (int. 32 bit)	U32 (unsigned int. 16 bit)
F4 (floating point 4 byte)	F (IEEE 8 byte floating point)
STR (string)	STR0T (NULL terminated C-style string)

return <newoffs> the offset (current fpos) after the write.

note

The <value> is converted to row bytes following <format> and <conversion>.
The <file> is opened, the file pointer is moved to <offset>, the row bytes are written. If -1 is specified for <offset>, <value> will be appended to the end of the file.
If the file does not exists it will be created.
<newoffs> can be used as <offset> for the next ~WRITEB().
See Appendix D for the file name.

example1:

```
~WRITEB("Def.tlg", 1561, 270, "II", "S16")
```
The following bytes will be written to the file: 0Eh 01h at offset 1561 (01h * 100h + 0Eh = 10Eh = 270).

example2:

```
~LET [fpos] = 0
~LET [line] = ~STRCAT("@echo off",~CHR(13),~CHR(10))
~LET [fpos] = ~WRITEB("ASCTEST.BAT”, [fpos], [line], "-", "STR")
~LET [line] = ~STRCAT("echo Hello World",~CHR(13),~CHR(10))
~LET [fpos] = ~WRITEB("ASCTEST.BAT”, [fpos], [line], "-", "STR")
~LET [line] = ~STRCAT("pause",~CHR(13),~CHR(10))
~LET [fpos] = ~WRITEB("ASCTEST.BAT”, [fpos], [line], "-", "STR")
```

command: ~READB

File Read Binary

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~WRITEB

syntax

~READB(<file>, <offset>, <qtbyte>, <format>, <conversion>)

input

<file> file that will be read in Binary mode
<offset> position (in bytes) inside the file (fpos)
<qtbyte> how many bytes
<format> see ~WRITEB
<conversion> all the ~WRITEB conversion operators except STR0T.

return the read-in value or string.

note

The <file> is opened, the file pointer is moved to <offset>, <qtbyte> bytes are readin from the file. This row data is interpreted following <format> and <conversion>.
See Appendix D for the file name.

example1

```
~LET [A] = ~READB("Def.tlg", 1561, 2, "II", "S16")
```
If the file contains the sequence 0Eh 01h at offset 1561, the value 010Eh = 270 is read-in. Will be [A] = 270.

example2

```
~LET [str] = ~READB("C:\Autoexec.bat", 0, 2, "-", "STR")
```
The read-in value will be the first 2 characters from the file.

command: ~READLINE

File Read in Text mode

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~READB, ~WRITEB

syntax

~LET [line] = ~READLINE(<file>, <line no.>)

input

<file>	file that will be read in text mode
<line no.>	the line number that has to be read-in. 1 is the 1st line.

return the requested line from the file.

note

The <file> is opened and scanned until <line no.> is read-in. If the line is empty but exists, the [CHR13] character is returned. If the line is not empty, all the characters excluding [CHR13] will be returned.
If the line does not exists, an empty line [CHR0] is returned, this can be used to check for EndOfFile.
If the file could not be opened, an error is returned.
See Appendix D for the file name.
The line that can be readin has a maximum length limit that depends by the decoder internal buffer capacity. If the file line is longer, it will be truncated.

example

```
~LET [l] = 0
~DO
  ~LET [l] = ~I([l] + 1)
  ~LET [str] = ~READLINE([file], [l])
~LOOP UNTIL([str] != [CHR0])
~MSGBOX("~READLINE",0,[file]," has ",[l]," lines")
```

command: ~FEXIST

Check for the file existence.

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~FEXIST(<file>)

input the file name that has to be checked.
return 0 if the file exists.
note

See Appendix D for the file name.

example

```
~LET [err] = ~FEXIST("def.tlg")
~IF [err] != 0
  ~MSGBOX(~MSG(error_msg),0,("def.tlg",~CHR(13),~MSG(file_not_found))
  ~EXIT_SUB
~ENDIF
~IF [err] = 0
  ~MSGBOX("FILE", 0, "The file exists.")
~ENDIF
```

command: ~FSIZE

Reads the file size.

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FEXIST

syntax

~FSIZE(<file>)

input the file name that has to be checked.
return the file size.
note

If the file was not foound, no error will pop-up.   See Appendix D for the file name.

example

```
~LET [err] = ~FEXIST("def.tlg")
~IF [err] != 0
  ~LET [fs] = ~FSIZE("def.tlg")
~ENDIF
```

command: ~COPYFILE
File copy

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~COPYFILE(<SourceFile>, <DestFile>)

input
<SourceFile> => the file that needs to be copied
<DestFile> => the file that needs to be created/updated.
note
See Appendix D for the file name.

example
```
~COPYFILE("def.tlg", "myatr.tlg")
```

command: ~DELFILE
Delete a file

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~DELFILE(<file>)

input the file name that has to be deleted.
return See Appendix D for the file name.

example
```
~DELFILE("myatr.tlg")
```

command: ~MKDIRS
Create directory

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CWD, ~ChDir

syntax
~MKDIRS(<isafile>, <path>)

input
<isafile> => 1 if <path> contains the file name. 0 if not.
<path> => the path that has to be created.
note
<path> can can contain (or not) the file name. If it has, <isafile> must be set to 1
The specified directory will be created. If it already exists, no error message is showed.
If the path contains more subdirectories, all of them will be created.
See Appendix D for the path/file name.

example
```
~LET [prev_dir] = ~CWD()
~MKDIRS(1, "C:\MyPrograms\door1.pgm")
~ChDir("C:\MyPrograms\door1.pgm")
...
~ChDir([prev_dir])
```

command: ~DELTREE
Esase one or more directory (folder)

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CWD, ~ChDir

syntax
~DELTREE(<path>)

input <path> => the path(s) that has(have) to be deleted. WildCards allowed.
return *0 on success.*

note
If the path contains more subdirectories, all of them will be erased.

See Appendix D for the path/file name.

example
```
~DELTREE("C:\TEMPdata")
```

command: ~CWD
Get the Current Working Directory

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~MKDIRS, ~ChDir

syntax
~CWD()

return *the Current Working Directory*

example
```
~LET [prev_dir] = ~CWD()
```

command: ~ChDir
Change the Current Working Directory

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CWD, ~MKDIRS

syntax
~ChDir(<path>)

input <path> => the path that has to be set as current.
return *0: Error 1: Ok*

note See Appendix D for the path/file name.

example
```
~LET [prev_dir] = ~CWD()
~MKDIRS(1, "C:\MyPrograms\door1.pgm")
~ChDir("C:\MyPrograms\door1.pgm")
...
~ChDir([prev_dir])
```

command: ~FILEFIND

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~DIRFIND

syntax
~FILEFIND(<file>, <find_id>)

input
```
<file>    := the file name that has to be found; WildCards are allowed in <file>.
<find_id> := search type: 0     => first search
                          not 0 => search continuation
```
return the found file name or "" at end: no (more) files.

note
* Search for all the matching files specified in <file>.
* Returns a file name each time it is called.
* The first time it is called, <find_id> must be 0.
* The next times it must be not 0 (better to use a variable intially set to 0 and then increment it).
* The search is dome when "" is back.
* Never change <file> between 2 calls.
* It is allowd to leave 2+ search open, because <file> is used to discriminate the correct one.
* A variable whose name is [ffh_<file>] is used to store intermediate data.
* An error is showed if <find_id> <> 0 but the Search did never begin.

See Appendix D for the path/file name.

example

```
~LET [qt_found] = 0
~DO
  ~IF(~FILEFIND("*.atr", [qt_found]) = "")
    ~EXIT LOOP
  ~LET [qt_found] = ~I([qt_found] + 1)
~LOOP
~MSGBOX("FILE SEARCH", ~STRCAT("Found ", [qt_found], " files(s)"))
```

command: ~DIRFIND

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~FILEFIND

syntax

~DIRFIND(<file>, <find_id>)

input

<file>   := the dir name that has to be found; WildCards are allowed in <file>.
<find_id> := search type: 0   => first search
                    not 0 => search continuation

return The found file name or "" at end: no (more) files.

note

See the notes for ~FILEFIND.   See Appendix D for the path/file name.

example   See the example for ~FILEFIND.

command: ~CompletePath

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~LET [fullname] = ~CompletePath(<Relative File Name>)

input <Relative File Name> := a valid File name
return: The file name with full path

command: ~Splitpath

Splits a path name into its components

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~SplitPath(<SourcePath>, "path", "fname", "ext")

input

<SourcePath>	Source path to be splitted
"path"	Name of the variable where the path has to be stored
"fname"	Name of the variable where the file name has to be stored
"ext"	Name of the variable where the file extension has to be stored

command: ~ZIP

Splits a path name into its components

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~ZIP(<Zip file>, <file to be zipped1>, …)

input

<Zip file>   := the ZIP file that will contain the Zipped files
<file to be zipped1> := file to be zipped, wildcards allowed.

note

If <Zip file> does not exist, it is created. If it exists it is updated and the new files will be added. If a file with the same name already exists in the Zip file, it is overwritten.
You can specify a path for both <Zip file> and the files to be zipped. If a path is not specified, the current one will be used. This function uses ZIP32.DLL Copyright (c) 1990-1999 Info-ZIP.

example

~ZIP("AllPrograms.zip", "*.pgm", "*.xxl")

command: ~EXECUTE

Run an operating system shell command.

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~EXECUTE(<style>, <program>, <cmdline>)

input

<style> The Operating System Window style
<program> The command that has to be run.
<cmdline> The command parameters.

return The Return code from the called <program>.

note

Valid style values can be found in the Operating System SDK
Help: 0 => Hide window (SW_HIDE)
1 => Normal Window (SW_NORMAL)
3 => Maximized Window (SW_SHOWMAXIMIZED)
The working directory for the program is the current directory: - the same .PCS Decoder path.

example

```
~EXECUTE(1, "C:\Xilog3\Xiso\Winxiso.exe", "C:\Xilog3\Pgm\a.xxl")
~LET [<retcode>] = ~EXECUTE(1, "C:\Xilog3\Xiso\Winxiso.exe", "a.xxl")
```

command: ~PUTDEF

Writes a DEF variable

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~GETDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS

syntax

~PUTDEF(<varname>, <value>)
~<DefHandle>.PutDef(<varname>, <value>)
~GenDefs.PutDef(<varname>, <value>)
~CfgDefs.PutDef(<varname>, <value>)
~DrwDefs.PutDef(<varname>, <value>)

input

<DefHandle>:= the handle to the default file
<varname> : the DEF variable that needs to be written
<value>: the value to be assigned to the DEF variable

note

If the variable does not exist, it is created. If it already exists, its is changed.

example1:

```
~LET [dhi] = ~LOADDEFS("FileIn.txt")
~LET [dho] = ~LOADDEFS("FileOut.txt")
~LET [myVar] = ~[dhi].GetDef("cargo_a")
~[dho].PutDef("cargo_a",[myVar])
~[dhi].CLOSEDEFS()
~[dho].CLOSEDEFS()
```

example2:

```
~PUTDEF("cargo_a", [A])
```

example3:

```
~GenDefs.PutDef("BigIcon", 1)
```

command: ~GETDEF, ~_GETDEF

Read a DEF variable

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PUTDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS

syntax

~GETDEF(<varname>[, <default value>])
~<DefHandle>.GetDef(<varname>[, <default value>])
~GenDefs.GetDef(<varname>)
~CfgDefs.GetDef(<varname>)

~DrwDefs.GetDef(<varname>)

input
<DefHandle> := the handle to the default file
<varname> := The DEF variable that needs to be retrieved.
<default value> := The value that has to be returned if the variable does not exists.
return The retrieved variable value.

note
If the variable does not exist, an error message is generated by this function.
Use: ~_GETDEF(<varname>) or ~GenDefs._GetDef(<varname>) ... to know if the variable exists: a 0 means that it exists (no errors).
If <default value> is specified and <varname> does not exists, <varname> is created and assigned the <default value>.

example1:
```
~LET [dhi] = ~LOADDEFS("FileIn.txt")
~LET [dho] = ~LOADDEFS("FileOut.txt")
~LET [myVar] = ~[dhi].GetDef("cargo_a")
~[dho].PutDef("cargo_a",[myVar])
~[dhi].CLOSEDEFS()
~[dho].CLOSEDEFS()
```
example4:
```
~LET [A] = ~ GetDef ("cargo_a", "12.34")
```
example2:
```
~LET [A] = ~GETDEF("cargo_a")
```
example3:
```
~IF ~_GetDef("cargo_a") == 0
  ~LET [A] = ~ GetDef ("cargo_a")
~ENDIF
```
example5:
```
~LET [A] = ~CfgDefs.GetDef("Numerical.Controller")
```

command: ~LOADDEFS
Load the DEF variables reading them from file

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~SAVEDEFS, ~CLOSEDEFS, ~PUTDEF, ~GETDEF, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS

syntax
~LET [dh] = ~LOADDEFS([<filename>][,<attributes>])

input <filename> := The file to be read-in. If not specified a default name is assigned.
<attributes> := An integer value that can be the sum of the following values. If not specified, the default value is 128 (128 = Read/Write + ASC_MODE + LOCKWRITE).
Allowed values:

1 Read	0 Write	0 Read/Write	4 PUT_AND_WRITE
8 NO_SUBDIRS	16 SHAROOT	32 BIN_MODE	0 ASC_MODE
64 LOCKREAD	128 LOCKWRITE	512 MUSTEXIST	1024 CREATE

return The handle to the created DEF class.

note If the attributes are not specified, the default ones are used:
If the file does not exist, it is created. The file is locked.

example 1:
```
~LOADDEFS()
```
example2:
```
~LET [dhi] = ~LOADDEFS("FileIn.txt",513)
~LET [dho] = ~LOADDEFS("FileOut.txt")
~LET [myVar] = ~[dhi].GetDef("cargo_a")
~[dho].PutDef("cargo_a",[myVar])
~[dhi].CLOSEDEFS()
~[dho].CLOSEDEFS()
```

command: ~SAVEDEFS
Save the DEF variables on to file

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CLOSEDEFS, ~LOADDEFS, ~PUTDEF, ~GETDEF, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS, ~RMDEFS

syntax
~SAVEDEFS([<filename>])

input <filename> := The file name for saving the Def variables.
note
If <filename> is not specified, the default is taken. The file remains locked.

Do not use this syntax if you Put/Get using the <DefHandle> syntax.

example
~SAVEDEFS()
~SAVEDEFS("MyDoor1")

command: ~CLOSEDEFS
Close the DEF variables file

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~SAVEDEFS, ~LOADDEFS, ~PUTDEF, ~GETDEF, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS, ~RMDEFS

syntax
~CLOSEDEFS([<save>])
~<DefHandle>.CLOSEDEFS([<save>])

input
<DefHandle> := the handle to the default file
<save> := 0 discard any changes.
1 save the changes to the file before closing it (default parameter).
return The name (complete with path) of the just closed file.

note
If <save> is not specified, the default value is taken. The file is unlocked.
example
~CLOSEDEFS()
~CLOSEDEFS(0)
~[dh].CLOSEDEFS(0)

command: ~PUSHCHDEFS, ~_PUSHCHDEFS
Change the file section and save the last one

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PUTDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS ~RMDEFS

syntax
~PUSHCHDEFS(<SectionName>)
~_PUSHCHDEFS(<SectionName>)
~<DefHandle>.PUSHCHDEFS(<SectionName>)
~<DefHandle>._PUSHCHDEFS(<SectionName>)

input
<SectionName> := The name of the section.
note
If the file section does not exist, an error message is generated by ~PUSHCHDEFS().
Use ~_PUSHCHDEFS(<SectionName>) to know if the SectionName exists: a return of 0 means that it exists (no errors).

example1:
```
~LET [RET] = ~PUSHCHDEFS("Section_1")
```

example2:
```
~IF ~_PUSHCHDEFS("Section_1") == 0
  ~LET [RET] = ~PUSHCHDEFS("Section_1")
  ~LET [RET] = ~GETDEF("Section_1_variable_a")
  ~POPDEFS()
~ENDIF
```

command: ~POPDEFS ~_POPDEFS
restore the previous DEF section

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PUTDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~MKDEFS, ~CHDEFS, ~RMDEFS

syntax
~POPDEFS()
~_POPDEFS()
~<DefHandle>.POPDEFS()

~<DefHandle>._POPDEFS()

note Use this command with the ~PUSHCHDEFS command.

example
```
~IF ~_PUSHCHDEFS("Section_1") == 0
  ~LET [RET] = ~PUSHCHDEFS("Section_1")
  ~LET [RET] = ~GETDEF("Section_1_variable_a")
  ~POPDEFS()
~ENDIF
```

command: ~MKDEFS, ~MKDEFS

Create a file section

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PUTDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~CHDEFS, ~RMDEFS

syntax

~MKDEFS(<SectionName>)
~_MKDEFS(<SectionName>)
~<DefHandle>MKDEFS(<SectionName>)
~<DefHandle>_MKDEFS(<SectionName>)

input

<SectionName> := The name of the section.

example
```
~LET [RET] = ~MKDEFS("Section_1")
```

command: ~CHDEFS, ~_CHDEFS

Go into a file section

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PUTDEF, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~RMDEFS

syntax

~CHDEFS(<SectionName>)
~_CHDEFS(<SectionName>)
~<DefHandle>.CHDEFS(<SectionName>)
~<DefHandle>._CHDEFS(<SectionName>)

input

<DefHandle> := the handle to the default file
<SectionName> := The name of the section.

note

If the file section does not exist, an error message is generated.
Use: ~_PUSHCHDEFS(<SectionName>) or ~_CHDEFS(<SectionName>) Or ~[…]._CHDEFS(<SectionName>) to know if the SectionName exists: a 0 means that it exists.

example1:
```
~LET [RET] = ~CHDEFS("Section_1")
```

example2:
```
~LET [dhi] = ~LOADDEFS("FileIn.txt")
~[dhi].CHDEFS("Section_1")
~LET [myVar] = ~[dhi].GetDef("cargo_a")
~[dhi].CLOSEDEFS()
```

example3:
```
~IF ~_CHDEFS("Section_1") == 0
  # We are now in the section
  #~LET [RET] = ~CHDEFS("Section_1")
  ~LET [RET] = ~GetDef("cargo_a")
~ELSE
  ~MSGBOX(0,"ERROR","Section [","Section_1", "] does not exists")
~ENDIF
```

command: ~RMDEFS, ~_RMDEFS

remove a DEF section including all of its content; the other sections are left unchanged

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PutDef, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS, ~CHDEFS

syntax

~RMDEFS(<SectionName>)

~_RMDEFS(<SectionName>)
~<DefHandle>.RMDEFS(<SectionName>)
~<DefHandle>._RMDEFS(<SectionName>)

input
<DefHandle> := the handle to the default file
<SectionName> := The name of the section.

note
You can not delete the current DEF section. If the file section does not exist, an error message is generated. Use: ~_PUSHCHDEFS(<SectionName>) or ~_CHDEFS(<SectionName>) Or ~[…]._CHDEFS(<SectionName>) to know if the SectionName exists: a 0 means that it exists.

example1:
```
~LET [RET] = ~RMDEFS("Section_1")
```
example2:
```
~LET [dhi] = ~LOADDEFS("FileIn.txt")
~[dhi].RMDEFS("Section_1")
~[dhi].CLOSEDEFS()
```
example4:
```
# delete the section without checking if it already exists:
~_RMDEF("Section_1")
```
example3:
```
~IF ~_CHDEFS("Section_1") == 0
  # We are now in the section
  ~CHDEFS("ROOT")
  ~RMDEF("Section_1")
~ELSE
  ~MSGBOX(0,"ERROR","Section [","Section_1", "] does
   not exists")
~ENDIF
```

command: ~IniFileRead, ~IniFileWrite

Quick (block) variables read/write

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~PutDef, ~LOADDEFS, ~SAVEDEFS, ~CLOSEDEFS, ~PUSHCHDEFS, ~POPDEFS, ~MKDEFS

syntax
~IniFileRead(<IniFileName>, <SECTION>, <global_prefix>)
~IniFileWrite(<IniFileName>, <SECTION>)

input
<IniFileName> := the ini file to be read-in/written. If the extension is not specified, .ini is assumed.
<SECTION> := file section, if not specified ROOT is assumed
<global_prefix> := variables name prefix for global variables; the other variables are treated as local.
"" : all the variables are local (default); "*" : all the variables are global

note <global_prefix> only applies to MACROS.

example1:
```
~IniFileRead("MyINI.ini",,"g_")
…
~IniFileWrite("MyINI.ini")
```
example2:
```
~IniFileRead("MyINI.ini",,"*")
~LET [VERSION] = ~I([VERSION]+1)
~IniFileWrite("MyINI.ini")
```

command: ~ShiftPoint

Shift a Point in 3D space

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~ShiftPoint(<X>,<Y>,<Z>,<Module>,<AngXY>,<AngXZ>)

input
<X> := the name of the variabile for the X coordinate of the point
<Y> := the name of the variabile for the Y coordinate of the point
<Z> := the name of the variabile for the Z coordinate of the point
<Module> := module for the shifting vector
<AngXY> := angle in the XY plane of the shifting vector
<AngXZ> := angle in the XZ plane of the shifting vector

note:
The function computer the shifting of a point from a position (x0, y0, z0) to a new position (x1, y1, z1).
The coordinate system to use this function can be described as follows:

- Imagine to observe a clock
- The origin of The X and Y axes is located in the clock's center.
- The positive direction of X axis is toward the 3 o clock
- The positive direction of Y axis is toward the 6 o clock

- Z axis grown toward us exiting from the clock
- AngXY means the angle (in DEGREE) on the XY plane, considerino 0 in the X axes and 90 in the Y,
- growing in clowise sense
- AngXZ means the angle (in DEGREE) on the XZ plane, considerino 0 in the XY plane, between the 3 and the 6 o clock, and 90 in the Z axes, in the positive sense

command: ~Get3DStepInfos

Get the machining data from an entity

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~Get3DStepInfos(<Idx>,<isA>,<PL>,<CP>,<tool>,<criteria>,<x>,<y>,<z>,<aXY>,<aTilt>)

input

<idx > := the cad index of the entity
<isA > := the type of the entity
<PL> := from 0 to 1: the fraction of the entity where to do the computation
<CP> := controlled point
"B" to have the bit position, "P" to have the PIVOT position, "C" to have the spindle support position
<tool> := the tool to use to work the entity
<criteria> := The way to search the information

"A" the Safety position for the selected point	*"B" the Pre-Approach position for the selected point*
"C" the Approach position for the selected point	*"D" the position of the material entry point*
"E" the point in the depth	

.. a value to have the distance from the entry point outside the panel

Note the user can ask for example "ABCE" to have multiple information

Output:

<x> := an array of the x value for the requested point positions
<y> := an array of the y value for the requested point positions
<z> := an array of the z value for the requested point positions
<aXY> := an XY angle for the point at the <PL> position of the entity
<atilt> := an Tilt angle for the point at the <PL> position of the entity

return The number of the point returned in <x>, <y>, <z>

example

```
~LET [qtStart] = ~Get3DStepInfos(~I([i]),[isA],0,"B","101","ABCDE", "x","y","z","aXY","aTilt")

 ~LET [angxy_s] = [aXY]
 ~LET [angTilt_s] = [aTilt]
 ~FOR [counterS] = 0 TO [qtStart]-1
   ~LET [x_s] = ~Ary("x",[counterS])
   ~LET [y_s] = ~Ary("y",[counterS])
   ~LET [z_s] = ~Ary("z",[counterS])
   ~LET [angInZ] = ~([angTilt_s]+90.0)
   ~LET [MYANG] = [angInZ]
   ~CALL ANGGIRO
   ~LET [angInZ] = [MYANG]
 ~NEXT
```

command: ~OBJGET

retrieves the object properties

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~OBJGET(<isA>, <idx>, <property 1>, <variable for property 1>[, ..., <property N>, <variable for property N>])

input

<isA> := the object type
<idx> := the object index
<property 1> := the name of the requested property
<variable for property 1> := the variable name that will hold the property
…
<property N> := the name of the requested property
<variable for property N> := the variable name that will hold the property

note

You can specify as many property as will stay in the line and the maximum number of command line allowed

parameters (tipically you can ask for up to 7 properties).
Refer to the **Object properties** appendix for a list of the available properties.

example
```
~OBJGET([isA], [idx1], ".first", "idx")
```

command: ~ExtractParams

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~ExtractParams(<str>,<param delimiter>,<token delimiter>,<acceptEmpty>,<Params>)

input

<str>	:= the string to parse to search the parameters
<param delimiter>	:= the character to separate the parameters
<token delimiter>	:= the character to get parameter containg the param delimiter
<acceptEmpty>	:= 1 get also empty parameters; := 0 get only not empty parameters
<Parameters>	:= the name of the array used for the parameters

return: The number of the parameter in the string

example
```
~LET [STR] = ,,,,10,,,20,30,"40",50,60,70 80,"hello, to",all,"    my,friends"
~LET [TokDelim] = "
~LET [ParDelim] = ,
~LET [nParameters] = ~ExtractParams([STR],[TokDelim],[ParDelim],0,"Parameters")
~FOR [i] = 0 TO [nParameters]-1 STEP 1
  ~MSGBOX(,,-~Ary("Parameters", [i])-)
~NEXT
```

output:
```
-10-
-20-
-30-
-40-
-50-
-60-
-70 80-
-hello, to-
-all-
    -     my,friends –
```

command: ~ASC

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CHR, ~MID, Appendix C

syntax
~ASC(<character>)

input A character or a string.
return The ASCII code of the character (the first string character).

example
```
  ~LET [ASCODE] = ~ASC("A")
```

command: ~CHR

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~ASC, Appendix C

syntax
~CHR(<ascii code>)

input *A valid ASCII code.*
return The corresponding character

example
~LET [LINEFEED] = ~CHR(13)

command: ~STRCAT
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~STRCAT(<str>, ...)

input The strings or expressions that must be concatenated.
return The resulting string.

example
~LET [AandB] = ~STRCAT(" A + B = ", ~([A] + [B]))

command: ~MID
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~MID(<str>, <from>, <len>)

input
<str> := the source string
<from> := character position in <str>
(1 = first character)
<len> := the number of characters to get. if not specified, all the remaining characters will be get.
return Picks a piece of <str>.

note Returns "" if nothing was picked.

example
~MID("ONE 2 THREE FOUR", 5, 1) => picks "2"

command: ~LEFT
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~ASC, ~RIGHT, ~MID, ~STRCAT

syntax
~LEFT(<str>, <len>)

input
<str> := the source string
<len> := the number of characters to get.
if not specified, the whole string will be get.
return Picks the first <len> string characters from <str>.

note Returns "" if nothing was picked.

example
~LEFT("ONE 2 THREE FOUR", 3) => picks "ONE"

command: ~RIGHT
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~RIGHT(<str>, <len>)

input
<str> := the source string
<len> := the number of characters to get. if not specified, the whole string will be get.
return Picks the lastt <len> string characters from <str>.

note Returns "" if nothing was picked.

example
```
~RIGHT("ONE 2 THREE FOUR", 4) => picks "FOUR"
```

command: ~INSTR
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~INSTR(<str>, <substr>)

input
<str> := the source string, containing the data to be found.
<substr> := the string that has to be found in <str>
return The position of <substr> inside <str>.

note Returns 0 if not found.

example
```
~LET [pos] = ~INSTR("ONE 2 THREE FOUR", "2")
# now [pos] = 5
```

command: ~LEN
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~LEN(<str>)

input *The string.*
return The string length (i.e. the number of characters in the string).

example
```
~LET [L] = ~LEN("home")
# now [L] = 4
```

command: ~TRIM
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~TRIM(<str>[,<op>])

input
<str> := the string to be modified.
<op> := operation to be performed: {L, R or B}
return Removes any SPACE and TAB characters from <str>.
<op> = L or <op> = l
Removes the leading SPACEs and TABs from <str>
<op> = R or <op> = r
Removes the trailing SPACEs and TABs from <str>
<op> = B or <op> = b or <op> not specified
Removes both trailing and leading SPACEs and TABs from <str>

note
This function only returns the result without changing the source <str>.

example
```
~LET [SOURCE] = "  XXXY "
~LET [AFTER]  = ~TRIM([SOURCE], "L")
# now [AFTER] = "XXXY "
```

command: ~CASE
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax:
~CASE(<str>[,<op>])

input
<str> := the string to be modified.
<op> := operation to be performed: {L, U or 1}
return
Change the <str> characters case according to the <op> rules.

<op> = L or <op> = l
Sets the characters to Lower case
<op> = U or <op> = u
Sets the characters to Upper case
<op> = 1
Set the 1st character of each word to uppercase; all the others to lowercase.

note This function only returns the result without changing the source <str>.

example
```
~LET [SOURCE] = " XXx Yes"
~LET [AFTER1]  = ~ CASE([SOURCE], "L")
# now [AFTER1] = "xxx yes"
~LET [AFTER2]  = ~ CASE([SOURCE], "U")
# now [AFTER2] = "XXX YES"
~LET [AFTER3]  = ~ CASE([SOURCE], "1")
# now [AFTER3] = "Xxx Yes"
```

command: ~Compare
String manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax:
~Compare(<method>, <string1>, <string2>)

input
<method> := "s[<C>[<N>]]" string compare

"*i*" integer compare (discard decimals)	"*f*[<P>]" floating-point compare
<C>=*i* (Case insensitive) or *s* (Case Sensitive)	<N>= Number of chars to be compared
<P>= Compare precision for equality	

return

-1 : <string1> < <string2>	0 : <string1> = <string2>	1 : <string1> > <string2>

<op> = L or <op> = l
Sets the characters to Lower case
<op> = U or <op> = u
Sets the characters to Upper case
<op> = 1
Set the 1st character of each word to uppercase; all the others to lowercase.

examples:
```
~Compare("si",    "Joe",  "JOE")  #returns 0
~Compare("ss2",   "Joe",  "JOE")  #returns 1
~Compare("i",     "14.3", "14.5") #returns 0
~Compare("f1E-1", "14.3", "14.5") #returns -1
```

command: ~SETSTATUS
Progress window update

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~SETSTATUS(<int_val>, <str_val>)

input
<int_val> => integer value
<str_val> => string value
note
C++ development (internal use only):
The meaning of the parameters depends on the Program interpretation of those values.

See SetStatusFn() in Common\Decoder.dll\Extdec.cpp

ccommand: **~dimAry ~boundsAry ~delAry ~Ary ~LETARY ~RENARY ~COPYARY**

Array manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~dimAry(<array variable>, <start index>[, <array size>])
~boundsAry(<array variable>, ["start index"], ["end index"], ["array size"])
~Ary(<array variable>, <index>)
~LETARY(<array variable>, <index>) = <expression>
~delAry(<array variable>)
~RENARY (<new array variable>, <array variable>)
~COPYARY(<destination array variable>, <source array variable>)

input

<array variable> := The array variable name
<start index> := index for the 1st element (usually 0 or 1)
<end index> := index for the last element
<array size> := number of elements in the Array. If not specified there is no limit.
<index> := index of the element to be accessed.
Must be within <start index> and <start index> + <array size> - 1

~boundsAry() *return*

The array bounds: index for the first element, index for the last element, number of elements (=<start index> + <array size> - 1)
The specified variables are filled-in with the requested values.
See the example.

~Ary() *Returns* The <index>-nd element or END if the requested element was not found.

note

* Warning, remember to type ~Ary() and not ~ary(). ~ary() (the 1st character is lower-case)is a different command.
* If you do not specify <array size>, the array dimension is automatic.
In that case you have one limitation: the ~LETARY() commands must be called using contiguous <index> values starting from <start index> (you can not call ~LETARY("DRIVES",1) if ~LETARY("DRIVES",0) has not been called, and so on...)
* If you try to read-in an element that has not been defined yet, you get "".

example 1

```
  ~dimAry("DRIVES", 0, 4)
  ~LETARY("DRIVES", 0) = "A:"
  # note: element 1 has not been written
  ~LETARY("DRIVES", 2) = "C:"
  ~LETARY("DRIVES", 3) = "D:"
  ~LET [test] = Drive letters are: ~Ary("DRIVES",
0),~Ary("DRIVES", 2), ~Ary("DRIVES", 3)
  # The [test] variable is now = "Drive letters are: A:,C:,D:"
  ~boundsAry("DRIVES", "from",, "size")
  # The [from] variable is now = 0
  # The [size] variable is now = 4
```

example 2 (automatic dimension):

```
  ~dimAry("DRIVES", 0)
  ~LETARY("DRIVES", 0) = "A:"
  ~LETARY("DRIVES", 1) = "C:"
  ~LETARY("DRIVES", 2) = "D:"
  ~LET [test] = Drive letters are: ~Ary("DRIVES",
0),~Ary("DRIVES", 1), ~Ary("DRIVES", 2)
  # The [test] variable is now = "Drive letters are: A:,C:,D:"
```

NOTE for C++ programmers:
The Array can be created/written/read-in from the C++ Source code this way:
Functions prototypes:

```
int Decoder::Dim_Array(char* arrayName, int fromIdx, int size = -1);
int Decoder::GetBounds_Array(char* arrayName, int *from, int *to, int *size);
int Decoder::Del_Array(char* arrayName);
int Decoder::Write_Array(char* arrayName, char* val, int idx);
int Decoder::Read_Array(char* arrayName, char **val, int idx);
```

example

```
#include "common\decoder.dll\array.h"
...
int  i, grf_asc_size = 3;
char *grf_asc[3];

grf_asc[0] = "A:";
grf_asc[1] = "C:";
grf_asc[2] = "D:";

decoder->ArrayList->Dim("GRF_ASC", 0, grf_asc_size);
```

```
for(i = 0; i < grf_asc_size; ++i)
  decoder->ArrayList->Write("GRF_ASC", grf_asc[i], i);
decoder->AddCamVar("GRF_ASC_SIZE", grf_asc_size);
```

command: ~ary

Array manipulation

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~dimAry, ~LETARY, ~Ary, ~DelAry

syntax

***~LET [element]* = ~ary(<array variable>, <index>)**

Warning: ~ary() is different from ~Ary(), see notes.

input

<array variable> := list of N variables, separated by ","
<index>   := 0 ... N - 1

return The <index>-nd element of the array or END if the requested element was not found.

note

Warning: **old syntax, no more to be used**. Use the ~dimAry(), ~LETARY() and ~Ary() functions instead.
~ary() and ~Ary() are 2 fully different functions (the 1st character is lower-case here).

example

```
~LET [idxary] = "12,15,9"
~LET [idx] = ~ary([idxary], 0)
# Now [idx] is 12
```

command: ~Struct

PRELIMINARY, SUBJECT TO MODIFICATIONS

Define a structure

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~Dims, ~Sname

syntax:

~STRUCT(“<STRUCT NAME>”, “<PAR1>”,...)

input

<STRUCT NAME> := the name (between brackets) of the new structure followed by the structure elements names

note

This function defines a new structure. See ~Dims() for further details.
It is a good practice to add a trailing _s to the structure name so you can easily recognize it.

example

```
~STRUCT(“MyLine_s”, “xs”, “ys”, “xe”, “ye”) # define the structure
~DIMS(“line”, “MyLine_s”)    # create a variable of type structure
~LET [line].xs = 10
~LET [line].ys = 20
~LET [line].xe = 100
~LET [line].ye = 200
~MSGBOX(,,”Line data: (“, [line].xs,” “,[line].ys,”)-(“, [line].xe,” “,[line].ye,”)”)
```

command: ~Dims

PRELIMINARY, SUBJECT TO MODIFICATIONS

Create a variable of type structure

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~Struct, ~Sname

syntax:

~Dims(“<VAR NAME>”, “<STRUCT NAME>”)

input

<VAR NAME>   := the name (between brackets) of the new variable of type structure

<STRUCT NAME> := the name (between brackets) of the structure

note
This function crates a new variable of type structure. You will be able to read and write each element in a way similar to variables. See the example.
You can assign structure values to array elements. Each array element can hold a different structure type variable. You can copy entire structures using the ~LET command.

example1:
~STRUCT("MyLine_s", "xs", "ys", "xe", "ye") # define the structure
~DIMS("line", "MyLine_s") # create a variable of type structure
~LET [line].xs = 10
~LET [line].ys = 20
~LET [line].xe = 100
~LET [line].ye = 200
~MSGBOX(,,"Line data: (", [line].xs," ",[line].ys,")-(", [line].xe," ",[line].ye,")")

example2:
~STRUCT("MyLine_s", "xs", "ys", "xe", "ye") # define the structure
~DIMS("tmp_line", "MyLine_s") # create a variable of type structure
~LET [tmp_line].xs = 10
~LET [tmp_line].ys = 20
~LET [tmp_line].xe = 100
~LET [tmp_line].ye = 200
~LET myArray(0) = [tmp_line]
~MSGBOX(,,"Line data: (", [line].xs," ",[line].ys,")-(", [line].xe," ",[line].ye,")")

example3:
~STRUCT("MyLine_s", "xs", "ys", "xe", "ye") # define the structure
~DIMS("my_line", "MyLine_s") # create a variable of type structure
~LET [tmp_line] = [my_line]

command: ~Sname

PRELIMINARY, SUBJECT TO MODIFICATIONS

Returns the structure type name of the structure variable

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~Struct, ~Dims

syntax:
~Sname("<VAR NAME>")

input
<VAR NAME> := the name (between brackets) of the variable whose structure type is to be known
return The structure type as defined in the ~STRUCT() command

note
Useful if you assign structure values to array elements and each array element can hold a different structure type variable.

example
~STRUCT("MyLine_s", "xs", "ys", "xe", "ye") # define the structure
~DIMS("line", "MyLine_s") # create a variable of type structure
~MSGBOX(,,"variable line is a struct of type: ", ~SNAME("line"))

command: ~CfgGet

Get configuration axis information data

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CfgPut, ~CfgWrite

syntax
~CfgGet(<info>)

input
<info> *:= "axis**N**min" or "axis**N**max" or "axis**N**park" or "axis**N**maxs" or "axis**N**acc" or "axis**N**dec" or "axis**N**rot"*
***N** = 1,2,3………… (1 = X,2 = Y,3 = Z….)*
return The requested value from the Configuration.

note This function reads from the configuration the requested data and returns it.

example
~LET [AXIS_Y_MAX] = ~CfgGet(*axis2max*)

command: ~CfgPut

Put a configuration axis information data

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CfgGet, ~CfgWrite

syntax

~CfgPut(<info>,<value>)

input

<info1> := *"axis**N**min" or "axis**N**max" or "axis**N**park" or axis**N**maxs" or*
*"axis**N**acc" or "axis**N**dec" or "axis**N**rot"*
***N** = 1,2,3………… (1 = X,2 = Y,3 = Z….)*

<value> := *the new value*

note

This function writes to the configuration the specified data.
You will need to call ~CfgWrite in order to effectively save the information.

example

```
~LET [AXIS_Y_MAX] = 3000.0
~CfgPut(axis2min, [AXIS_Y_MIN])
~CfgPut(axis2max, [AXIS_Y_MAX])
~CfgWrite()
```

command: ~CfgWrite

Save the modified configuration data

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~CfgPut

syntax

~CfgWrite()

note saves the information modified by ~CfgPut.

example

```
~LET [AXIS_Y_MAX] = 3000.0
~CfgPut(axis2min, [AXIS_Y_MIN])
~CfgPut(axis2max, [AXIS_Y_MAX])
~CfgWrite()
```

command: ~Atr

Gets the tools property

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~LET [TOOL_DATA] = ~Atr("T=<tool number>", <Requested Field>)
~LET [TOOL_DATA] = ~Atr("TC=<tool code>", <Requested Field>)
~LET [TOOL_DATA] = ~Atr(<tool number>, <Requested Field>)

input

<tool number> := *the number of the tool (ex. 10, 101, . . .)*
<Requested Field> := one of the following:

"ANGLE"	Asks for the Tool angle (tapered tool)
"BITTYPE"	Asks for the Bit Type. ~Atr()returns **P**, **L** or **S**
"DIAM"	Asks for the Tool Diameter (Disc Saw: width)
"DIAM_C_HEIGHT"	Asks for the Tool Cone height (tapered tool)
"DIAM_ING"	Asks for the Tool minimum Diameter (tapered or profiled tool)
"DIAM_MAX"	Asks for the Tool minimum Diameter (tapered or profiled tool)
"DIAM_MIN"	Asks for the Tool minimum Diameter (tapered or profiled tool)
"EXIST"	~Atr() returns 1 if the tool exists
"ROT_DIR"	Asks for the Tool Rotation dir. ~Atr()returns **CW** or **CCW**
"SVAS_HEIGHT"	Asks for the Tool Countersink height
"T_ANG"	Asks for the Tool horizontal angle (vector angle)
"T_LENGTH"	Asks for the Tool useful length (Disc Saw: useful radius)
"T_LENGTH_TOT"	Asks for the Tool total length (Disc Saw: radius)
"T_PMAG"	Asks for the Tool configuration stock number

"T_RADIUS"	Asks for the Tool radius
"T_X"	Asks for the Tool configuration X position
"T_Y"	Asks for the Tool configuration Y position
"T_Z"	Asks for the Tool configuration Z position
"T_ZANG"	Asks for the Tool vertical angle (tilting angle)
“FACE”	Tool working side "UPPER", "LEFT", "RIGHT", "FRONT", "LOWER" or "VECTOR"
"TOOL_CODE"	
"TOOL_DESCR"	
"TTYPE"	Asks for the Tool Type. ~Atr() returns: TTYPE_NONE or TTYPE_BIT or TTYPE_ROUTER or TTYPE_SAW or TTYPE_CONIC or TTYPE_SENSOR or TTYPE_SPECIAL.
"TYPE"	Asks for the Tool Type. ~Atr() returns: 0 (No tool) or 1(Cylindrical router) or 2 (Profiled router) or 3 (Tapered router) or 4 (Saw) or 5 (special or sensor router);
"RPM_MAX"	Asks for the Tool rotation maximum speed
"RPM"	Asks for the Tool rotation speed
"START_FEED"	Asks for the Tool start feed
"FEED"	Asks for the Tool path feed
"MAX_FEED"	Asks for the Tool maximum execution feed
“TOOL_AGR_IDX”	Aggregate index for multi tool tools
“WORKING_DIR”	Asks for the Working direction

return The value of the requested data.

example
~LET [tdiam] = ~Atr("101", "DIAM")

command: ~SetTool

Set the machining data to an entity

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~SetTool(<isA variable>, <idx variable>, <tool data>)

input
<isA variable> *:= the type of the entity*
<idx variable> *:= the index of entity*
<tool data> := a string with the machining data

example
~SetTool(“IS_SHAPE”, 0, “T=102,PRY=1,F=1000”)
~SetTool(“IS_SHAPE”, 0, “TC=T10,PRY=1,F=1000”)

command: ~GetTool

Get the machining data from an entity

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax
~GetTool(<isA variable>, <idx variable>, <use tool code>)

input
<isA variable> *:= the type of the entity*
<idx variable> *:= the index of entity*
<use tool code> := 1 to get the machining data with the tool code
0 to get the machining data with the tool number

return: the string with the machining data

example
~LET [MACHININGDATA] = ~GetTool(“IS_SHAPE”, 0, 0)
~LET [MACHININGDATA] = ~GetTool(“IS_SHAPE”, 0, 1)

Dialog commands

command: ~EXTRA

Dialog Window definition

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~INPUT, ~_INPUT, ~INPUT_OK

syntax

~EXTRA <extraname> (~)[,<extra title>[, <extra bitmap>]]
<variable name>,<message>,<variable type>
[,<default value>[,<X pos>,<Y pos>[,<X size>,<Y size>[,<Callback fn>]]]]
...
~END

input

<extraname> := the Dialog box name to be used for retrieving
<extra title> := the title for the Dialog Box
<extra bitmap> := the Bitmap to be showed in the Dialog Box. Optional.

<variable name> := the variable to be associated with the field
<message> := the message to be displayed
<variable type>

C => string	D => floating point	M => floating point with automatic Millimeter/Inch conversion
I => integer	B => boolean (checkbox)	L => label

<default value> := the default value to be associated to the variable the first time the Dialog is run (see notes). Optional.
<X pos>,<Y pos> := the edit box position. Optional.
<X size>,<Y size> := the edit box size. Optional.
<Callback fn> := the function to be called for event processing. Optional.

note

- Warning: no spaces must be inserted before <variable name>.
- The first time the dialog is run, the <default value> is used; after having pressed the Ok button the changed value is stored on to a file and retrieved the next time the dialog will pop-up.
  If required, the value of any variable can be forced to any desired value ignoring the stored one. See the ~INPUT command line parameters for further details.

example

```
~EXTRA DOORARC_DLG (~), ~MSG(3), macroply.bmp
[D], ~MSG(4), M, 33
[1], ~MSG(5), L
[2], ~MSG(9), L
~END

~START TEST
  ~INPUT DOORARC_DLG
~END
```

command: ~INPUT, ~_INPUT, ~INPUT_OK

Open a Dialog Window

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~EXTRA

syntax

~INPUT <extraname>[, <extra variable value>,...]
~_INPUT <extraname>[(<extra variable value>,...)]
~INPUT_OK <extraname>[, <extra variable value>,...]

input

<extraname> := The EXTRA Dialog Window definition
<extra variable value> := The value to be forced. Can be an expression.

return

1 if the *Ok* button has been pressed
0 if the *Cancel* button has been pressed (Warning!: see notes).

note

- <extra variable value> can be specified in order to force the value for the ~EXTRA fields.
- ~INPUT and ~_INPUT open up a dialog Window, and initializes the variables if Ok has been

pressed.
~INPUT_OK does not pop-up, only initializes the variables specified in the ~EXTRA section.

- The difference between ~INPUT and ~_INPUT is that if you press the Cancel button, ~INPUT stops executing the Processor, ~_INPUT continues the execution and returns 0 so the programmer can decide what to do.

example

```
~EXTRA DOORARC_DLG (~), ~MSG(3), macroply.bmp
[vD], ~MSG(4), M, 33
[v1], ~MSG(5), L
[v2], ~MSG(9), L
~END
~START TEST1
  # [vD], [v1] and [v2] will be initialized as specified
  # in the ~EXTRA section.
  # The dialog will pop-up.
  ~INPUT DOORARC_DLG
  # now [vD], [v1] and [v2] have been initialized.
~END
~START TEST2
  # [vD] will be forced to 34
  # [v1] will be forced to 2
  # The dialog will pop-up.
  ~INPUT DOORARC_DLG (34, 2)
  # now [vD], [v1] and [v2] have been initialized.
~END
~START TEST3
  # [vD], [v1] and [v2] will be initialized as specified
  # in the ~EXTRA section.
  # The dialog will NOT pop-up.
  ~INPUT_OK DOORARC_DLG
  # now [vD], [v1] and [v2] have been initialized.
~END
~START TEST4
  # [vD], [v1] and [v2] will be initialized as specified
  # in the ~EXTRA section.
  # The dialog will pop-up.
  ~LET [ret] = ~_INPUT DOORARC_DLG
  ~IF [ret] = 1
    # now [vD], [v1] and [v2] have been initialized.
    ~MSGBOX(,,"User pressed Ok")
  ~ELSE
    # [vD], [v1] and [v2] have NOT been initialized.
    ~MSGBOX(,,"User pressed Cancel")
  ~ENDIF
~END
```

command: ~DlgBegin

Create a Dialog Window

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~DlgBegin(<parentWin>, <title>, <dim x>, <dim y>)

input

<parentWin>	:= The Dialog ID of the parent Window, or nothing if this is the first Window.
<title>	:= Window title.
<dim x>	:= Window X dimension
<dim y>	:= Window Y dimension

return The Dialog ID of the created Window.

example

```
~LET [Dlg] = ~DlgBegin(, "Title for my Window", 150, 70)
```

command: ~DlgItem

Add a control to a Window

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~DlgItemProperty()

syntax

~DlgItem(<DlgID>, "Label", <ID>, <text>, <x>, <y>, <dx>, <dy>, <style>)
~DlgItem(<DlgID>, "EditBox", <ID>, <variable>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
~DlgItem(<DlgID>, "IntegerBox", <ID>, <variable>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
~DlgItem(<DlgID>, "DoubleBox", <ID>, <variable>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
~DlgItem(<DlgID>, "MminBox", <ID>, <variable>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
**~DlgItem(<DlgID>, "CheckBox", <ID>, <text>, <variable>, <x>, <y>, <dx>, <dy>,
<CallBackFn>,<style>)**
**~DlgItem(<DlgID>, "RadioButton", <ID>, <text>, <variable>, <x>, <y>, <dx>, <dy>,
<CallBackFn>,<style>)**
~DlgItem(<DlgID>, "Button", <ID>, <text>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
**~DlgItem(<DlgID>, "BmpButton", <ID>, <text>, <bmp>, <x>, <y>, <dx>, <dy>,
<CallBackFn>,<style>,<TextPosition>)**
~DlgItem(<DlgID>, "Bmp", <ID>, <bmp>, <x>, <y>, <dx>, <dy>)

~DlgItem(<DlgID>, "ListBox", <ID>, <SelItem>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
~DlgItem(<DlgID>, "ComboBox", <ID>, <SelItem>, <x>, <y>, <dx>, <dy>, <CallBackFn>,<style>)
~DlgItem(<DlgID>, "Avi", <ID>, <AviName>, <x>, <y>, <dx>, <dy>)

input

<DlgID> := The Window who owns this control.

<ID> := unique control identifier for each ~DlgItem.
For Windows standard controls use the following predefined IDs:

1 IDOK	2 IDCANCEL	3 IDABORT
4 IDRETRY	5 IDIGNORE	6 IDYES
7 IDNO	8 IDCLOSE	9 IDHELP

<variable> := Is the name of the variable that is associated with the control.
It is the name of a Processor variable or array element:
- "PANG" associates the variable [PANG] of the processor to the control.
- "A(19)" attaches element 19 of the array A of the processor to the control. You can set the initial value with ~LETARY("A",19) = … and will be able to retrieve the value with ~Ary("A",19). The array must be dimensioned first.

<style> := [ws_val] | [ws_val] |...

WS_CHILD	= 40000000h	ES_AUTOVSCROLL	= 0040h
WS_VISIBLE	= 10000000h	ES_AUTOHSCROLL	= 0080h
WS_BORDER	= 00800000h	ES_LEFT	= 0000h
WS_TABSTOP	= 00010000h	ES_CENTER	= 0001h
WS_VSCROLL	= 00200000h	ES_RIGHT	= 0002h
WS_HSCROLL	= 00100000h	ES_MULTILINE	= 0004h
SS_LEFT	= 00000000h (for Label only)	ES_READONLY	= 0800h
SS_CENTER	= 00000001h (for Label only)	ES_WANTRETURN	= 1000h
SS_RIGHT	= 00000002h (for Label only)	BS_AUTOCHECKBOX	= 00000003h (for CheckBox only)
BS_AUTORADIOBUTTON	= 00000009h (for RadioButton only)	BS_PUSHBUTTON	= 00000000h (for Button and BmpButton only)
LBS_NOTIFY	= 0001h (for ListBox only)	LBS_SORT	= 0002h (for ListBox only)
CBS_DROPDOWNLIST	= 0003h (for ComboBox only)		

<TextPosition> := The text position relative to the bitmap
Top = 2 Bottom = 3 Left = 4 Right = 5 Inside = 6

<AviName> := Name of the video clip to show

return differs from 0 if the control has been created.

Default <style> values:

Label:	WS_CHILD|WS_VISIBLE|SS_LEFT
EditBox:	WS_CHILD|WS_VISIBLE| WS_BORDER|WS_TABSTOP| ES_AUTOHSCROLL|ES_LEFT
IntegerBox:	WS_CHILD|WS_VISIBLE|WS_BORDER|WS_TABSTOP| ES_AUTOHSCROLL|ES_LEFT
DoubleBox:	WS_CHILD|WS_VISIBLE|WS_BORDER|WS_TABSTOP| ES_AUTOHSCROLL|ES_LEFT
MminBox:	WS_CHILD|WS_VISIBLE|WS_BORDER|WS_TABSTOP| ES_AUTOHSCROLL|ES_LEFT
CheckBox:	WS_CHILD|WS_VISIBLE|WS_TABSTOP|BS_AUTOCHECKBOX
RadioButton:	WS_CHILD|WS_VISIBLE|WS_TABSTOP|BS_AUTORADIOBUTTON
Button:	WS_CHILD|WS_VISIBLE|WS_TABSTOP|BS_PUSHBUTTON
BmpButton:	WS_CHILD|WS_VISIBLE|WS_TABSTOP|BS_PUSHBUTTON
Bmp:	WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON
ListBox:	WS_TABSTOP|WS_BORDER|LBS_NOTIFY|LBS_SORT|WS_VSCROLL| WS_CHILD|WS_VISIBLE
ComboBox:	WS_CHILD|WS_VISIBLE|WS_TABSTOP|CBS_DROPDOWNLIST

example

```
~DlgItem([dlg], "Bmp", 112, "Logo.bmp", 5, 6, 100, 70)

to create a multiline editbox:
<style> = WS_BORDER|WS_TABSTOP|ES_AUTOHSCROLL|ES_LEFT| ES_MULTILINE|ES_WANTRETURN
        = 00800000h|00010000h|0080h|0000h|0004h|1000h = 811084h
        811084h -> 8458372 (integer value)
~DlgItem([dlg],"EditBox",103,"desr",10,60,150,45,,"8458372")
```

command: ~DlgItemProperty

Change a control property in a Window

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~DlgItem()

syntax

~DlgItemProperty(<DlgID>, <ID>, <Property Name>[, <Value>])
*** Object "ListBox" and object "ComboBox" have the following properties:**
~DlgItemProperty(<DlgID>, <ID>, "Add", <Value>[, <HiddenValue>])
~DlgItemProperty(<DlgID>, <ID>, "SelGet")
~DlgItemProperty(<DlgID>, <ID>, "SelGetH")
~DlgItemProperty(<DlgID>, <ID>, "SelGetPos")
~DlgItemProperty(<DlgID>, <ID>, "SelPut", <Value>[, <HiddenValue>])
~DlgItemProperty(<DlgID>, <ID>, "Del", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "DelPos", <Position>)
~DlgItemProperty(<DlgID>, <ID>, "GetPos", <Position>)
~DlgItemProperty(<DlgID>, <ID>, "Sel", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "SelPos", <Position>)
~DlgItemProperty(<DlgID>, <ID>, "Selh", <HiddenValue>)
*** "Del" and “DelPos” return the number of remaining items after Deletion.**
*** Object "Button" has the following properties:**
~DlgItemProperty(<DlgID>, <ID>, "SetText", <Value>)
*** Object "BmpButton" has the following properties:**
~DlgItemProperty(<DlgID>, <ID>, "SetBmp", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "SetText", <Value>)
*** Object "Bmp" has the following properties:**
~DlgItemProperty(<DlgID>, <ID>, "SetBmp", <Value>)
*** All objects have the following properties:**
~DlgItemProperty(<DlgID>, <ID>, "Show", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "Hide", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "SetGray", <Value>)
~DlgItemProperty(<DlgID>, <ID>, "SetTextColor", <Red>,<Green>,<Blue>)
~DlgItemProperty(<DlgID>, <ID>, "SetBackGroundColor", <Red>,<Green>,<Blue>)
~DlgItemProperty(<DlgID>, <ID>, "Tip", <TipText>)

input

<DlgID>	:= The Window who owns this control.
<ItemType>	:= The control type.
<ID>	:= Control unique identifier (ID).
<Property Name>	:=
<Value>	:=
<HiddenValue>	:= User Hidden string associated with the selected field. Can be read-in by calling "SelGetH".
<Position>	:= the position in the list (0 based index)
..<TipText>	:= Text displayed in the control’s tool tip

return The asked value or "" if the property has been set.

example

```
...
~DlgItem([dlg], "ListBox", 113, "Logo.bmp", 5, 6, 100, 70)
~DlgItemProperty([dlg], 113, "Add", "Element 1")
~DlgItemProperty([dlg], 113, "Add", "Element 2")
~DlgItemProperty([dlg], 113, "Add", "Element 3")
~DlgItemProperty([dlg], 113, "Sel", "Element 2")
~DlgExecute([dlg])
~LET [selected] = ~DlgItemProperty([dlg], 113, "SelGet")
```

command: ~DlgExecute

Run a Dialog Window

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~DlgExecute(<DlgID>)

input

<DlgID> := The Window who owns this control.

return The value associated with the Pressed button:

1 IDOK	2 IDCANCEL	3 IDABORT
4 IDRETRY	5 IDIGNORE	6 IDYES
7 IDNO	8 IDCLOSE	9 IDHELP

note

Each time an event occurs or any button is pressed, the Dialog values are saved in the relative variables. If you need to handle a Cancel event, you need to have 2 set of variables and restore the original values at the End if Cancel is pressed.

example

```
~LET [dlg_ret] = ~DlgExecute([dlg])
```

command: ~DlgCommand

Force the Window closure.

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~DlgCommand(<DlgID>[, <command>])

input

<DlgID> := The Window that has to be closed.
<command> :=

1	IDOK	2	IDCANCEL

notes *If* <command> is not specified, IDOK will be generated

example

```
~DlgCommand([dlg])
```

command: ~DlgEnd

Deallocates a closed Window.

applied to:	post-processor	macro	drw rebuild
	X	X	X

syntax

~DlgEnd(<DlgID>)

input

<DlgID> := The Window that has to be deallocated.
<command>

1	IDOK	0	IDCANCEL

note Run this command only after the ~DlgExecute() has completed for the given <DlgID>.

example

```
~DlgEnd([dlg])
```

command: ~TimeGet

Get the current time.

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~Time, ~TimeElapsed

syntax

~LET <tmData> = ~TimeGet()

return *The current time in a binary non-readable format,* ~Time() is required.

example

```
~LET [tmData] = ~TimeGet()
~LET [HH] = ~Time([tmData], "HOUR")
~LET [MM] = ~Time([tmData], "MIN")
~MSGBOX(,,"The Time is ",[HH], ":", [MM])
```

command: ~Time

Convert the time to a readable format.

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~TimeGet, ~TimeElapsed

syntax

~Time([<tmData>], <what>)

input
<what> := {“YEAR”,”MONTH”,”WDAY”,”DAY”,”HOUR”,”MIN”,”SEC”,”milliSEC”}
return *The requested value from <tmData>. If <tmData> is not specified the current time is used.*
note

YEAR	*returns the current year*	*MONTH*	*returns 1 (January) to 12 (December)*
WDAY	*returns 0 (Sunday), 1 (Monday) to 6 (Saturday)*	*DAY*	*return 1 to 31*
HOUR	*returns the current hour*	*MIN*	*returns the current minutes*
SEC	*returns the current seconds*	*milliSEC*	*returns the current milliseconds*

Warning: if you need to make subsequent calls to ~Time(), always first get the current time with ~TimeGet() and then get the data or you may get wrong data: when you make the second call the time has changed.

example1:
```
~LET [YEAR] = ~Time(, "YEAR")
~MSGBOX(,,"The current Year is ",[YEAR])
```

example2:
```
~LET [tmData] = ~TimeGet()
~LET [HH] = ~Time([tmData], "HOUR")
~LET [MM] = ~Time([tmData], "MIN")
~MSGBOX(,,"The Time is ",[HH], ":", [MM])
```

command: ~TimeElapsed

Computes the difference between 2 time references.

applied to:	post-processor	macro	drw rebuild
	X	X	X

See also: ~TimeGet, ~Time

syntax
LET [elapsed] = ~TimeElapsed(<tmData1>[,<tmData2>])

input
<tmData1> := the 1st time reference (equivalent to Chrono Start)
<tmData2> := the 2nd time reference (equivalent to Chrono Stop).
If not specified current time is computed
return *The time difference in milliseconds between the 2 time references.*

example
```
~LET [tmDataSTART] = ~TimeGet()
~Execute(“DIR”)
~LET [elapsed] = ~TimeElapsed([tmDataSTART])
~MSGBOX(,,"ExecutionTime is ",[elapsed],” milliseconds”)
```

macro General commands

command: ~g_LET

Creates a Global Variable. From now on this variable will be shared among all submacros and the main macro

applied to:	post-processor	macro	drw rebuild
		X	

See also: ~LET, ~(), ~I(), ~STRCAT()

syntax
~global(<VarName>[,<VarValue>])
~global(<ArrayName>,<ArrayStartIdx>,<ArraySize>)

note
It is strongly recommended to precede the variable name with a "g_" to distinguish the variable type.

example
```
~global("g_OK")
~global("g_Array()",0, -1)
```

command: ~RUN, ~RUNCMD

Executes a macro from inside another macro

applied to:	post-processor	macro	drw rebuild
		X	

See also: ~CALL

syntax
~RUN(<Macro Name>, <Macro file>, <Macro Subroutine>,<DefName>[,"<varname 1>",...])
~RUNCMD(<Macro file>, <Macro Subroutine>,<DefName>[,"<varname 1>",...])

input
<Macro Name> := the macro directory; if not specified, the current will be specified
<Macro file> := the macro (.mac) file; if not specified, Main.mac will be used. In ~RUNCMD if the path is not specified, [PROGDIR] will be used as path.
<Macro Subroutine> := the macro Subroutine; if not specified, the whole macro will be run (first ~DIALOG, then ~DRAW). See "<varname 1>" Notes below.
<DefName> := the default section for ~PUTDEF and ~GETDEF.
"<varname 1>"... := the local variable to be shared with the <Macro Subroutine>.

return If in the called macro, the [return] variable exists, ~RUN will return its value.

note
- ~RUN calls a new Macro, ~CALL starts a subroutine within the same macro.
- <DefName> is the name of the Default section for storing the variables used by ~PUTDEF and ~GETDEF. Each macro has its own private macro section. Specifying "ROOT" as name, all the Macros will share the same default variables, so you will be able to define some Global Environment variables. At the end of the program execution, the variables will be saved in the .ini file belonging to the first macro. If <DefName> is not specified, the default variables will be discarded at the end of the macro execution.
- If one or more "<varname 1>" is specified and <Macro Subroutine> is not specified, the ~DIALOG will not be executed.
- You are entitled to make more .mac files in the same directory. You will be able to run main.mac only by the Macro menu. Anyway you can run any .mac file by using ~RUN.

example
```
~RUN("MyMacroDir", "MyMacFile", "S1_sub", "ROOT", "a")
~LET [MyMacFile_res] = ~RUN(, "MyMacFile")
```

command: ~G0, ~MOVE
move ‘pen’ (prepare drawing)

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SetTool, ~DIAM, ~C, ~G1, ~G2, ~rG2, ~G3, ~rG3

syntax
~G0(<x>,<y>,<depth>,<tool diam>,<formulas>)
~MOVE(<x>,<y>,<depth>,<tool diam>,<formulas>)

input
<x> := the X coordinate of the start point
<y> := the Y coordinate of the start point
<depth> := the start depth
<tool diam> := the tool diameter or machining information;
see Appendix “Tool diameter and Machining parameters”

note
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.

command: ~G1, ~LINE
draws a line on the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~G0, ~alG1, ~G2, ~rG2, ~G3, ~rG3, ~G5, ~arG5

syntax
~LET [idx] = **~G1(<x>,<y>,<depth>,<formulas>)**
~LET [idx] = **~LINE(<x>,<y>,<depth>,<formulas>)**

input
<x> := the X coordinate of the end point
<y> := the Y coordinate of the end point
<depth> := the depth of the end point

return
The Graphic drawing handle, useful for changing it later if required

note

~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.

example
```
~G0(   0,   0,[DZ],20)
~G1([DX],   0,[DZ])
~G1([DX],[DY],[DZ])
~G1(   0,[DY],[DZ])
```

command: ~alG1, ~LINE_al
draws a line on the current face

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~G0, ~G1, ~G2, ~rG2, ~G3, ~rG3, ~G5, ~arG5

syntax
~LET [idx] = **~alG1(<angle>,<length>,<depth>,<formulas>)**
~LET [idx] = **~LINE_al(<angle>,<length>,<depth>,<formulas>)**

input

<angle>	:= the angle (in degrees) of the segment that has to be created
<length>	:= the length of the segment that has to be created
<depth>	:= the depth of the end point

return
The Graphic drawing handle, useful for changing it later if required

note
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.

example
```
~G0(0,0,5,10,"FX=L FY=T")
~alG1(45,141,"FX=R FY=B")
~LET [idx] = ~G1([DX],[DY],0,"FX=R FY=B")
```

command: ~G2, ~ARC_CW
draws a clockwise arc (end, center) on the current face

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~G0, ~G1, ~rG2, ~G3, ~rG3, ~ARC_3P, ~G5, ~arG5

syntax
~LET [idx] = **~G2(<x>,<y>,<depth>,<xc>,<yc>,<zc>,<formulas>)**
~LET [idx] = **~ARC_CW(<x>,<y>,<depth>,<xc>,<yc>,<zc>,<formulas>)**

input

<x>	:= the X coordinate of the end point
<y>	:= the Y coordinate of the end point
<depth>	:= the depth of the end point
<xc>	:= the X coordinate of the center point
<yc>	:= the Y coordinate of the center point
<zc>	:= the depth of the centerpoint

return
The Graphic drawing handle, useful for changing it later if required

note
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If the center coordinates are not quite correct, they are automatically adjusted.

example
```
~G0( 0,100,1,12)
~G2(200,100,1,100,100)
```

command: ~rG2, ~ARC_rCW
draws a clockwise arc (end, radius) on the current face

applied to:	post-processor	macro	drw rebuild
		X	

See also: ~G0, ~G1, ~G2, ~G3, ~rG3, ~ARC_3P, ~G5, ~arG5

syntax
~LET [idx] = **~rG2(<x>,<y>,<depth>,<r>,<formulas>)**
~LET [idx] = **~ARC_rCW(<x>,<y>,<depth>,<r>,<formulas>)**

input

<x>	:= the X coordinate of the end point
<y>	:= the Y coordinate of the end point
<depth>	:= the depth of the end point
<r>	:= the arc radius

return
The Graphic drawing handle, useful for changing it later if required

note
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If the radius coordinate is incorrect, it is automatically adjusted.

example
```
~G0(  0,100,1, 12)
~rG2(200,100,1,100)
```

command: ~G3, ~ARC_CCW

draws a counter-clockwise arc (end, center) on the current face

applied to:	post-processor	macro	drw rebuild
		X	

See also: ~G0, ~G1, ~G2, ~rG2, ~rG3, ~ARC_3P, ~G5, ~arG5

syntax
~LET [idx] = **~G3(<x>,<y>,<depth>,<xc>,<yc>,<zc>,<formulas>)**
~LET [idx] = **~ARC_CCW(<x>,<y>,<depth>,<xc>,<yc>,<zc>,<formulas>)**

input

<x>	:= the X coordinate of the end point
<y>	:= the Y coordinate of the end point
<depth>	:= the depth of the end point
<xc>	:= the X coordinate of the center point
<yc>	:= the Y coordinate of the center point
<zc>	:= the depth of the centerpoint

return
The Graphic drawing handle, useful for changing it later if required

note
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If the center coordinates are not quite correct, they are automatically adjusted.

example
```
~G0(  0,100,1,12)
~G3(200,100,1,100,100)
```

command: ~rG3, ~ARC_rCCW

draws a counter-clockwise arc (end, radius) on the current face

applied to:	post-processor	macro	drw rebuild
		X	

See also: ~G0, ~G1, ~G2, ~rG2, ~G3, ~ARC_3P, ~G5, ~arG5

syntax
~LET [idx] = **~rG3(<x>,<y>,<depth>,<r>,<formulas>)**
~LET [idx] = **~ARC_rCCW(<x>,<y>,<depth>,<r>,<formulas>)**

input

<x>	:= the X coordinate of the end point

<y>   := the Y coordinate of the end point
<depth> := the depth of the end point
<r>   := the arc radius

return
The Graphic drawing handle, useful for changing it later if required

note
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If the radius coordinate is incorrect, it is automatically adjusted.

example
```
~G0(0,100,1,12)
~rG3(200,100,1,100)
```

command: ~ARC_3P

draws an arc (end, mid-point) on the current face

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~G0, ~G1, ~G2, ~rG2, ~G3, ~rG3, ~G5, ~arG5

syntax
~LET [idx] = **~ARC_3P(<x>,<y>,<depth>,<xp>,<yp>,<formulas>)**

input
<x>   := the X coordinate of the end point
<y>   := the Y coordinate of the end point
<depth> := the depth of the end point
<xp>   := the X coordinate of a point along the arc
<yp>   := the Y coordinate of a point along the arc

return The Graphic drawing handle, useful for changing it later if required

note
The mid-point (<xp>,<yp>) determines whether the arc is clockwise or counter-clockwise.
~G1, ~G2, ~G3, … can be mixed in any sequence, but a ~G0 at the beginning is required
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.

example
```
~G0(  0,100,1, 12)
~ARC_3P (200,100,1,100,200)
```

command: ~G5, ~ARC_TG

draws an arc tangent to the previous entity

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~G0, ~G1, ~G2, ~rG2, ~G3, ~ARC_3P, ~arG5

syntax
~LET [idx] = **~G5(<x>,<y>,<depth>,<formulas>)**
~LET [idx] = **~ARC_TG(<x>,<y>,<depth>,<formulas>)**

input
<x>   := the X coordinate of the end point
<y>   := the Y coordinate of the end point
<depth> := the depth of the end point

return The Graphic drawing handle, useful for changing it later if required

note
~G5 can be mixed in any way with the other drawing commands (~G1, ~G2, ~G3, …) but requires that a previous drawing command, different from ~G0, has already been issued: a ~G0 followed by a ~G5 is not allowed. A previous entity is required for getting the tangent angle.
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If the end point is aligned with the previous entity, a tangent line is generated.

example

```
~G0(0,100,1,12)
~G5(100,0,1)
```

command: **~arG5, ~ARC_arTG**

draws an arc tangent to the previous entity

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~G0, *~G1, ~G2, ~rG2, ~G3, ~ARC_3P, ~G5*

syntax

~LET [idx] = **~arG5(<ang>,<r>,<depth>,<formulas>)**
~LET [idx] = **~ARC_arTG(<ang >,<r>,<depth>,<formulas>)**

input

<ang> := the aperture of the arc in degrees.
A positive value means ClockWise arc,
a negative value means Counter-ClockWise arc.
<r> := the radius of the arc that has to be created
<depth> := the depth of the end point

return The Graphic drawing handle, useful for changing it later if required

note

~G5 can be mixed in any way with the other drawing commands (~G1, ~G2, ~G3, …) but requires that a previous drawing command, different from ~G0, has already been issued: a ~G0 followed by a ~G5 is not allowed. A previous entity is required for getting the tangent angle.
If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.
If <ang> = 180 a tangent line is generated.

example

```
~G0(0,0,1,12)
~G1(100,0,1)
~arG5(90,100,1)
```

command: **~B**

makes a hole on the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~BB, ~SetTool, ~DIAM

syntax

~LET [idx] = **~B(<x>,<y>,<depth>,<tool diam>,<formulas>)**

input

<x> := the X coordinate of the hole
<y> := the Y coordinate of the hole
<depth> := the depth of the hole
<tool diam> := the tool diameter
see Appendix “Tool diameter and Machining parameters”

return The Graphic drawing handle, useful for changing it later if required

note *If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.*

example

```
~B(100,100,1,12)
```

command: **~BB**

makes a hole barrier on the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~B, ~DIAM

syntax

~LET [idx] = **~BB(<type-alignment>,<xs>,<ys>,<xe>,<ye>,<z>,<tool diam>,**
<step_x>,<step_y>,<rep_x>,<rep_y>,
<Start_Formulas>,<End_Formulas>)

input
<type-alignment> := <T><A1><A2>
<T> := the type of Barrier:

<T>	**Mnemonic**	**Description**
A	Automatic (default)	Auto selects Vertical or Horizontal or Matrix depending on <xs>,<ys>,<xe>,<ye>
V	Vertical	
H	Horizontal	
M	Matrix	
D	Diagonal	

<A1>, <A2> := the Barrier alignment relative to the selection rectangle:

<A1>,<A2>	**Description**
S	Align 1st hole at the Start of barrier
C	Center holes (default)
E	Align last hole at the End of barrier
	If a type Matrix barrier is to be created, both <A1> and <A2> must be specified: <A1> is for Horizontal alignment, <A2> is for Vertical alignment. example ~BB("MCC",…).

<xs> := the X coordinate of the start point of the rectangle
<ys> := the Y coordinate of the start point of the rectangle
<xe> := the X coordinate of the end point of the rectangle
(if not specified = <xs>)
<ye> := the Y coordinate of the end point of the rectangle
(if not specified = <ys>)
<z> := the depth of the holes
<tool diam> := the tool diameter;
see Appendix "Tool diameter and Machining parameters"
<step_x> := the distance between holes (default is 32.00 mm)
<step_y> := the distance between holes (default is 32.00 mm)
<rep_x> := the number of horizontal holes
<rep_y> := the number of vertical holes
<Start_Formulas> := the (optional) formula for the rectangle Start point
<End_Formulas> := the (optional) formula for the rectangle End point

return The Graphic drawing handle, useful for changing it later if required

note *If a parameter is not specified, the latest one (same parameter from the latest instruction) is used.*

example
Make an horizontal holes row from (100,100) to (400,100), depth=5mm, diam=8mm:
~BB("HC",100,100,400,,5,8)
Make a vertical holes row from (100,100) to (100,400):
~BB("VC",100,100,,400,5,8)

command: ~DIAM

sets the default diameter value

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax
~DIAM(<tool diam>)

input
<tool diam> := the default tool diameter
see Appendix "Tool diameter and Machining parameters"

note *After ~DIAM has been set-up, you do not need to specify the* <tool diam> where required.

example
~DIAM(12)
~B(100,100,1)

command: ~C

sets the tool correction

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax

~C(<tool correction>)

input

<tool correction> := 0 Set Tool correction = none,
1 Set Tool correction = right,
2 Set Tool correction = left,
3 Set Tool correction = lenght

note *You must set this command before the working (~G0) begin.*

example

```
~C(1)
~G0(10,10)
~G1(100,100)
```

command: ~F

sets the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~F(<panel face>)

input

<panel face> := 1 (upper),
2 (right),
3 (left),
4 (front),
5 (rear),
6 (bottom)

example

```
~F(1)
```

command: ~DIM

draws a quote

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax for Horizontal dimension:

~DIM("H", <xs>,<ys>, <xe>,<ye>, <dtext>)

syntax for vertical dimension:

~DIM("V", <xs>,<ys>, <xe>,<ye>, <dtext>)

syntax for aligned dimension:

~DIM("AL",<xs>,<ys>, <xe>,<ye>, <dtext>)

syntax for radius dimension:

~DIM("R", <xc>,<yc>, <x>,<y>, <dtext>)

syntax for angle dimension (not available yet):

~DIM("A", <xc>,<yc>,<ang1>,<ang2>,<dtext>)

input

<xs>	:= the X coordinate of the 1st point of the quote
<ys>	:= the Y coordinate of the 1st point of the quote
<xe>	:= the X coordinate of the 2nd point of the quote
<ye>	:= the Y coordinate of the 2nd point of the quote
<xc>	:= the X coordinate of the center point of the quote
<yc>	:= the Y coordinate of the center point of the quote
<x>	:= the X coordinate of a point on the arc
<y>	:= the Y coordinate of a point on the arc
<dtext>	:= the text distance

example

```
# The following example dimensions the entire panel
~F(1)
~DIM("H",0,0,[DX],0,-15)
~DIM("V",0,0,0,[DY],-15)
~F(2)
~DIM("H",[DZ],0,[DZ],[DY],15)
~DIM("V",0,0,[DZ],0,-15)
~F(3)
~DIM("H",[DZ],0,[DZ],[DY],15)
~DIM("V",0,0,[DZ],0,-15)
~F(4)
~DIM("H",0,[DZ],[DX],[DZ],15)
~DIM("V",0,0,0,[DZ],-15)
~F(5)
~DIM("H",0,[DZ],[DX],[DZ],15)
~DIM("V",0,0,0,[DZ],-15)
```

command: ~SETPANEL

changes the panel size

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~SETPANEL(<DX>,<DY>,<DZ>,<use formulas>,<remove ext holes>)

input

<DX>	:= the new panel Horizontal size
<DY>	:= the new panel Vertical size
<DZ>	:= the new panel Width
<use formulas>	:= 1: apply formulas if any 0: do not apply formulas (default)
<remove ext holes>	:= 1: remove the holes outside the panel 0: do not remove any hole (default)

command: ~Offset

Creates an Offset path starting from the selected path

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~OffsetA

syntax for Left/Right offset:

~LET [ret] = **~Offset([<offset>], [<dir>]   [, <isa variable>, <idx_array variable>])**

syntax for Point offset:

~LET [ret] = **~Offset([<offset>], "P", <x>, <y>[, <isa variable>, <idx_array variable>])**

input

<offset>	:= distance, greater than 0 If not specified, the latest value will be used.
<dir>	:= "L" or "R" (Left or Right of the profile direction) If not specified, the latest value will be used.
<x>	:= the X coordinate of a point (see note)
<y>	:= the Y coordinate of a point (see note)
<isa variable>	:= variable that will be filled with the just created offset path type.
<idx_array variable>	:= variable that will be filled with the just created offset path idx(s). The result will be like this: <idx1>,...,<idxN>

return   The variables <isa variable> and <idx_array variable> will be filled with the resulting path.
Returns a value different from 0 if everything went ok.

note

A path must be selected before calling this command.
The Offset path will be created at the left or right of the original path. You can directly specify “L” for left or “R” for right or “P” and then a point. If the point is on the right the right offset will be generated. If the point is on the left the left offset will be generated. If the point is along the original path, an error will be generated.

example

```
~LET [res] = ~SELSINGLE("Please select the path you want to offset")
~IF [res] = 0
  ~EXIT_SUB
~ENDIF
~SELGETOBJ(0, "isA", "idx1")
# now the path has been selected
~LET [res] = ~Offset(100, "L", "isA", "idx2")
~IF [res] = 0
  ~EXIT_SUB
~ENDIF
```

command: ~OffsetA

Creates an Offset path starting from the selected path

applied to:	post-processor	macro	drw rebuild

☐	X	☐

See also: ~Offset

syntax

```
~OffsetA(<array name>, <offset>, <dir>, <corner_type>,
        <isa variable>, <idx_array variable>,
        <idx_index_array variable>,<qt_path_ret variable>,<RemoveLoop>])
~OffsetA(<array variable>,[<offset>], "P", <x>, <y>,[<corner_type>]
        [, <isa variable>, <idx_array variable>,
         <idx_index_array variable>,<qt_path_ret variable>,<RemoveLoop>])
```

input

```
  <array name>   := array containing the shapes index of which the
                    offset must be computed
  <offset>       := distance, greater than 0
  <dir>          := "L" or "R" (Left or Right of the profile direction)
  <corner_type>  := Offset corner type: 1 fillet type
                                         2 linear extension
                                         3 chamfer type
  <isa variable> := variable that will be filled with the just created
                    offset path type.
  <idx_array>    := array that will be filled with the just created
                    offset path(s). This array will be automatically created.
                    If an array with the same name already exists it will
                    be overwritten.
  <idx_array variable> variable that will be filled with the just created
                      offset path idx(s). The result will be like this:
                      <idx1>,...,<idxN>
  <idx_index_array variable> variable that will be filled with the indexes of
                            idx_array variable array representative the
                            start idx of each profile
  <qt_path_ret variable>     Quantity of created offseted profiles
  <RemoveLoop>               If TRUE make only external path (for intersecting
                            profiles). Default value : FALSE
```

return The number N of created entities (the *<idx_array>* array size).

example

```
  ~dimAry("Shapes_Array", 0, 4)
  …
  ~OffsetA("Shapes_Array",10.0,"R",0,"isA","Shapes_Array_offs")
```

command: **~TextSet**

Set the Text Font

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~Text

syntax

~TextSet(<font>, <sizex>, <sizey>, <alignment>, <extraspace>)

input

```
 <font>       the font file and not the font name, see Notes.
 <sizex>      the font size X
<sizey>       the font size Y
 <alignment>  := <H><V> where <H> := {'T', 'B', 'C'}
                              <V> := {'L', 'R', 'C'}
             examples: "TL" or "CC"
 <extraspace> extra spaces in mm between 2 characters
```

note

The <font> field must be filled with the font file and not the font name.
You can get it from the Advanced button in the Cad Text command. examples: Normal.slf, Arial.ttf

example

```
   ~TextSet("Normal.slf", 15, 16,"CC", 0)
   ~Text("Hello", ~(DX/2), ~(DY/2), 2, 4)
```

command: ~Text

Write text

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~TextSet

syntax

~Text(<text>,<x>,<y>,<z>,<diam>,<formulats>,<array shape>,<array hole>)

input

<text> := The text string
<x> := The text X initial position
<y> := The text Y initial position
<z> := The text Z initial position
<diam> := The tool diameter
<Formulas> :=
<array shape> : = returned array with the handle of the shapes created with the Text command
<array hole> : = returned array with the handle of the holes created with the Text command

example

```
~TextSet("Normal.slf", 15, "CC", 0)
~Text("Hello", ~(DX/2), ~(DY/2), 2, 4)
```

command: ~CreateBmp

Make a Bitmap starting from a view of the drawing

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~MergeBmp

syntax

**~CreateBmp(<new bitmap name>, <dimension x>, <dimension y>, <color>,
<rotation>, <mirror>, <exploded>)**

input

<new bitmap name> := the name of the resulting bitmap
<dimension x> := the X dimension in pixel of the resulting bitmap
<dimension y> := the Y dimension in pixel of the resulting bitmap
<color>

0 - black and white	1 - colors - default value	

<rotation>

0 no rotation - default value	1 rotation of 90 degrees	2 rotation of 180 degrees
3 rotation of 270 degrees		

<mirror>

0 - disable the mirroring of the bitmap	1 - enable the mirroring of the bitmap

<exploded>

0 - create the bitmap as the current display	1 - create the bitmap in the exploded view

example

```
~CreateBmp([NewbitmapName], [dimX], [dimY], 1, 0)
```

command: ~MergeBmp

Make a new Bitmap by merging 2 other bitmaps

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~CreateBmp

syntax

~MergeBmp(<bitmap1>, <bitmap2>, <new bitmap>, <position>)

input

<bitmap1> := the name of the first bitmap file.
<bitmap2> := the name of the second bitmap file.
<new bitmap name> := the name of the resulting bitmap file
<position> := the position of the second bitmap on the first one:

0 - upper left corner	1 - middle upper side	2 - upper right corner
3 - middle left side	4 - center (default value)	5 - middle right side
6 - bottom left corner	7 - middle bottom side	8 - bottom right corner

note *Size, color depth and all the other bitmap attributes will be get from <bitmap1>.*

example
```
  ~MergeBmp(“BackGround.bmp”, “Panel.bmp”, “Temp.bmp”, 3)
  ~MergeBmp(“Temp.bmp”, “Tool.bmp”, “ToBeDisplaied.bmp”, 3)
```

command: ~EditLabel

Show the ASPAN label profile editor to modify the layout of the labels

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~EditLabel (<.lba name>)

input

<.lba name> := the name of the .lba file for the label profile to show at the start of the label editor

return The name of the label profile the the user choose in the label editor

example
```
~LET [profileStart] = c:\\prof1.lba
~LET [profileEnd] = ~EditLabel([profileStart])
```

command: ~PrnLabels

Run a label printing process

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~PrnLabels(<drw names>, <.lba name>)

input

<drw names> := the name of the drw to print label for or the name of the array of drw names

<.lba name> := the name of the .lba file for the label profile

example
```
  ~dimAry("drawings",0,[nDrws])
  ~LETARY("drawings", 0) = C:\drw\1.drw
  ~LETARY("drawings", 1) = C:\drw\2.drw
   . . .      . . .
  ~LETARY("drawings", [nDrws]) = C:\drw\n1.drw
  ~PrnLabels("drawings()")
```

… print the labels for each drawing of the array with the default ASPAN label profile

```
  ~LET [drwname] = “C:\drw\1.drw”
  ~LET [lba] = “ C:\profiles\lba1.lba”
  ~PrnLabels([drwname], [lba])
```

… print the label for the single drawing with the ASPAN label profile loaded

command: ~ObjsJoin

Join 2 objs

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~ObjsJoin(<isA1>, <idx1>, <side1>, <isA2>, <idx2>, <side2>)

link <side1> of obj1 with <side2> of obj2

input

<side1>, *<side2>* = "S" (start) or "E" (end)

note

The 2 objs must be of the same type. Holes can not be linked.
The 2 objs must have the same Start/End point.
obj1 == obj2 is not allowed.
obj2 versus may sometimed be reversed in order to complete the operation.

command: ~ObjCut

Cut an obj by creating 2 distinct objects

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~ObjsJoin

syntax

~ObjCut(<isA>, <idx>[, <side>])

input

Cut obj by creating 2 distinct objects. The "Cut" point depends on:
<*side*> = "S" (start) or "E" (end). If not specified <side> = "S".

note

If the obj is already the first/last shape in a link, the cut operation makes non-sense. Anyway no error is showed.
This function is complementary to ~ObjsJoin().

command: ~ObjDel

Delete the specified Object

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax

~ObjDel(<isA>, <idx>[, <linked>])

input

<*isA*>, <*idx*> is object.
<*linked*> = 1 if the Object is linked to other entities, all the linked entities are deleted.
If not specified defaults to 0.

return The number of deleted Objects.

note

Deletes the Object(s).
If <linked> = 0 the object must not be linked to other entites.
If <linked> = 1 the object and all the linked entites are deleted.
If the object is selected, it is automatically unselected.

example

```
~ObjDel([isA], [idx])
```

command: ~Idx2Hnd

Get the handle of an entity starting from its index

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax

~Idx2Hnd(<isA>, <idx>)

input

<*isA*> := the type of the entity
<*idx*> := the index of the entity

return The handle of the entity

example

```
~LET [HND] = ~Idx2Hnd([isA], [idx])
```

command: ~Hnd2Idx

Get the index of an entity starting from its handle

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax

~ Hnd2Idx(<isA>, <hnd>)

input

<*isA*> := the type of the entity
<*hnd*> := the handle of the entity

return The index of the entity

example
~LET [IDX] = ~Hnd2Idx([isA], [hnd])

macro Selection commands

command: ~GETPOINT
Allows to capture a position in the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~GETSEGMENT

syntax
~GETPOINT("<varname X>", "<varname Y>", "<varname Z>", <message>)

input
<varname X> := Variable name that will have to be filled with the X value
<varname Y> := Variable name that will have to be filled with the Y value
<varname Z> := Variable name that will have to be filled with the Z value
<message> := Help message to be showed

return

1 if the point has been grabbed	0 if the point has not been grabbed

command: ~GETSEGMENT
Allows to capture 2 positions in the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~GETPOINT

syntax
~GETSEGMENT("<varname XS>", "<varname YS>", "<varname ZS>", "<varname XE>", "<varname YE>", "<varname ZE>", <message for 1st point>, <message for 2nd point>)

input

<varname XS>	Variable name that will have to be filled with the X value for 1st point
<varname YS>	Variable name that will have to be filled with the Y value for 1st point
<varname ZS>	Variable name that will have to be filled with the Z value for 1st point
<varname XE>	Variable name that will have to be filled with the X value for 2nd point
<varname YE>	Variable name that will have to be filled with the Y value for 2nd point
<varname ZE>	Variable name that will have to be filled with the Z value for 2nd point
<message for 1st point>	Help message to be showed for 1st point
<message for 2nd point>	Help message to be showed for 2nd point

command: ~SELSINGLE
Select 1 object in the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELMULTI, ~SELOBJS, ~SELGETOBJ

syntax
~LET [qt] = **~SELSINGLE([<message>])**

input
<message> := Help message to be showed

return The number of the selected objects

command: ~SELMULTI
Select + objects in the current face

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELSINGLE, ~SELOBJS, ~SELGETOBJ

syntax
~LET [qt] = **~SELMULTI([<message>])**

input
<message> = Help message to be showed

return The number of the selected objects

command: ~SELOBJS

Select all the objects inside the specified rectangle or around the specified point

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~SELSINGLE, ~SELGETOBJ, ~SELGETOBJ

syntax
~LET [qt] = **~SELOBJS(<type>, <x1>, <y1>[, <x2>, <y2>])**

input

<*type*>	"C" => Crossing selection	"W" => Window selection
<x1>	X coordinate of the 1st rectangle point	
<y1>	Y coordinate of the 1st rectangle point	
<x2>	X coordinate of the 2nd rectangle point	
<y2>	Y coordinate of the 2nd rectangle point	

return The number of the selected objects

note
If <x1>, <y1> only are specified all the entities around the specified point are selected.
If the Crossing selection is specified all the entities crossing the rectangle(or point) are selected.
If the Window selection is specified all the entities fully inside the rectangle are selected.

command: ~SELOBJ

Force the selection of 1 object

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~SELSINGLE, ~SELGETOBJ

syntax
~LET [qt] = **~SELOBJ(<isA>, <idx>[, <SelAdd>])**

input

<isA>	The type of the object that has to be selected
<idx>	The index of the object that has to be selected
<SelAdd>	

1 If other objects are already selected the selection is keeped	0 If other objects are already selected the selection is cleared

return The number of the objects selected by this command

command: ~UNSELOBJ

Force the deselection of 1 object

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

See also: ~SELSINGLE, ~SELGETOBJ, ~SELRECORD

syntax
~UNSELOBJ(<isA>, <idx>)

input

<isA>	The type of the object that has to be de-selected
<idx>	The index of the object that has to be de-selected

note
If the Object was not selected, nothing happens.
If the Object has linked objects, all are deselected.

Use the command **~SELRECORD(0,1)** to de-select everything.

example
~UNSELOBJ([isA], [idx])

command: **~SELRECORD**

Entities Selection/Deselection recording command

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELSINGLE, ~SELGETOBJ, ~UNSELOBJ

syntax
~SELRECORD(<rec>[, <clear>])

input
<rec>

1 => start recording: all the entities created from now on, will be selected (default)	0 => stop recording: all the entities created from now on, will no more be selected

<clear>

1 => If some entity is already selected, the previous selection will be lost	0 => If some entity is already selected, the previous selection will not be lost (default)

note Use the command **~SELRECORD(0,1)** to de-select everything.

command: **~SELGETOBJ**

retrieve the selected objs one by one

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELSINGLE, ~SEL_QT

syntax
~SELGETOBJ(<counter>, <isA variable>, <idx variable>)

input

<counter>	must be a number between 0 and the number of the selected objects
<isA variable>	The variable that has to be filled with the Obj type
<idx variable>	The variable that has to be filled with the Obj index

return The number of the returned objects (0 or 1)

note
You can get the number of the currently selected objects from the selection functions or from ~SEL_QT().

example
~SELGETOBJ(0, "isA", "idx")
~MSGBOX("~SELGETOBJ", 0, "The 1st selected entity is: ", [isA], "[", [idx], "]")

command: **~SELDELOBJ**

Delete all selected entites

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELGETOBJ, ~SEL_QT

syntax
~SELDELOBJ()

command: **~SEL_QT**

returns the number of the selected entities

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SELGETOBJ, ~SEL_QT

syntax
~LET [qt] = **~SEL_QT()**

return The number of the currently selected entities

command: ~OBJSET

changes the object properties

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~OBJGET, SELGETOBJ

syntax

~OBJSET(<isA>, <idx>, <property 1>, <value 1>[, ..., <property N>, <value N>])

input

<isA>	the object type
<idx>	the object index
<property 1>	the name of the property that has to be changed
<variable for property 1>	the value for the property
…	
<property N>	the name of the requested property
<variable for property N>	the variable name that will hold the property

note

You can specify as many property as will stay in the line and the maximum number of command line allowed parameters (tipically you can ask for up to 7 properties).
Please refer to the **Object properties** appendix for a list of the available properties.

example

```
~SELGETOBJ(0, "isA", "idx1")
# now the path has been selected
#
~OBJSET([isA], [idx1], ".diam", "12.0")
```

command: ~OBJS_QT

returns the number of objects

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = **~OBJS_QT(<isA>)**

input

<isA> the object type

return The number of objects of the specified type for the current drawing.

note The objects in all the faces are enumered.

macro CAM commands

command: ~EDGEBANDING

assign an edge banding operation to the specified entity

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = **~EDGEBANDING(<shape_index>, <edge_side>, <lead-in_speed>, <edge_speed>, <edge_number> [, <edge_height>, <edge_thickness>)**

input

<shape_index> :=
<edge_side> := edge banfing side
"R" - right; "L" - left; in according to the direction of the specified routing
<lead-in_speed> :=
<edge_speed> :=
<edge_number> := 0 – 6

only if you set the <edge_number> to 0, you have to specify the following values
<edge_height> :=
<edge_thickness> :=

return The edge banding handle.

example
```
~LET [res] = ~EDGEBANDING([index], "L", 1000, 5000, 1)
~LET [res] = ~EDGEBANDING([index], "L", 1000, 5000, 0, 23.00, 2.00)
```

command: ~EDGEBANDINGCHECKLENGHT

check the length of the entity to be edged

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = **~EDGEBANDINGCHECKLENGHT(<shape_index>, <edge_side)**

input

<shape_index> :=
<edge_side> := edge banfing side
"R" - right; "L" - left; in according to the direction of the specified routing

return

0 - If the length is correct.
The length value if the length is not enough

example
```
~LET [lenght] = ~EDGEBANDINGCHECKLENGHT([index], "Shape_Array")
```

command: ~EDGEBANDINGCHECKLENGHT

check the length of the entity to be edged

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = **~EDGEBANDINGCHECKLENGHT(<shape_index>, <edge_side>)**

input

<shape_index> :=
<edge_side> := edge banfing side
"R" - right; "L" - left; in according to the direction of the specified routing

return

0 - If the length is correct.
The length value if the length is not enough

example
```
~LET [lenght] = ~EDGEBANDINGCHECKLENGHT([index], "Shape_Array")
```

command: ~GETTANGENTPATH

get the list of entities that are linked with the selected entity and that compose the tangent path

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = **~GETTANGENTPATH(<shape_index>, <Array_name>)**

input

<shape_index> :=
<Array_name> := the name of the Array where the entities are stored

return The number of objects that compose the tanget path

example
```
~LET [qt] = ~GETTANGENTPATH([index], "Shape_Array")
```

command: ~ENDTRIMMING

get the list of entities that are linked with the selected entity and that compose the tangent path

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~ENDTRIMMING(<shape_index>, <tool_number>, <tool_correction>, <type>, <pass_number>, <offset>,<edge_side>)

input

<shape_index> :=
<tool_number> := the number of the saw tool (ex. 10, 101, . . .)
<tool_correction> := "R" – right ; "L" - left; "A" - automatic;

<type> := "C" – blade center; "E" – blade edge;
<pass_number> := 1; 2;
<offset> :=
<edge_side> := "I" – initial edge; "F" – final edge; "B" – both edges;

example

~LET [res] = ~ENDTRIMMING([index], 180, "R", "C", 1, 0)

macro Display commands

command: ~ToUserCoordinates

coordinates conversion

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~FromUserCoordinates

syntax

~ToUserCoordinates([<variable x>],[<variable y>],[<variable z>],[<variable ang>])

input

<variable x> := The name of the variable that contains the X coordinate
<variable y> := The name of the variable that contains the Y coordinate
<variable z> := The name of the variable that contains the Z coordinate
<variable ang> := The name of the variable that contains the angle

note

Convert coordinates from Aspan internal (origin at the top left corner) to user coordinates. Useful only if the coordinates need to be displayed.

example

~ToUserCoordinates("XI", "YI", "")
~INPUT DIALOG01([XI], [YI])
~FromUserCoordinates("XI", "YI", "")

command: ~FromUserCoordinates

coordinates conversion

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~ToUserCoordinates

syntax

~FromUserCoordinates([<variable x>],[<variable y>],[<variable z>],[<variable ang>])

input

<variable x> := The name of the variable that contains the X coordinate
<variable y> := The name of the variable that contains the Y coordinate
<variable z> := The name of the variable that contains the Z coordinate
<variable ang> := The name of the variable that contains the angle

note

Convert coordinates from user to Aspan internal (origin at the top left corner). Useful only if the coordinates have been Inputed from the user.

example

~ToUserCoordinates("XI", "YI", "")
~INPUT DIALOG01([XI], [YI])
~FromUserCoordinates("XI", "YI", "")

command: ~UpdateWin

force the screen update

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~ZoomExt

syntax

~UpdateWin

note
Useful if some computing operation is been performed but not completed but need to show that the Macro is still working.

command: ~ZoomExt

Updates the current view size

syntax
~ZoomExt()

example
~ZoomExt()

command: ~PickTool

Select a tool from current machine tooling

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax
~LET [tool] = **~PickTool(<entity type>, <tool number>)**

input
<entity type> *:= "IS_SHAPE" or "IS_HOLE"*
<tool number> *:= the number of the tool (ex. 10, 101, . . .)*

return The number of the chosen tool

note Pops up a dialog allowing the tool selection.

example
~LET [NEW_TOOL] = ~PickTool("IS_SHAPE", "101")
~LET [NEW_TOOL] = ~PickTool("IS_SHAPE", [OLD_TOOL])

command: ~SetTool

Set a tool to the specified entity

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~GetTool, ~G0, ~B

syntax
~SetTool(<isA>, <idx>, <tool data>)

input
<isA > := "IS_SHAPE" or "IS_HOLE"
<idx > := the entity idx
<tool data> := see Appendix "Tool diameter and Machining parameters"

example
~SetTool([isA], [idx], "T=101,SF=1000,F=4000")

command: ~GetTool

Get the tooling data for the specified entity

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~SetTool

syntax
~GetTool(<isA>, <idx>[, <Use Tool code>], <multimachining>])

input
<isA > := "IS_SHAPE" or "IS_HOLE"
<idx > := the entity idx
<Use Tool code> := 0 get the Tool Number (default)
1 get the Tool Code
<multimachining> := multimachining idx (default 0)

return A complete list of Tooling parameters.

note
see Appendix “Tool diameter and Machining parameters”

example1 (get the tooling info and the Tool number):
```
  ~GetTool([isA], [idx])
```
example2 (get the tooling info and the Tool Code):
```
  ~GetTool([isA], [idx], 1)
```
example3 (get the tooling info for the 2nd machinng):
```
  ~GetTool([isA], [idx], 0, 1)
```

command: **~Fillet**
create a fillet between two entities

applied to:	post-processor	macro	drw rebuild
	[ ]	[X]	[ ]

syntax
~Fillet(<radius>, <idx_first_entity>, <idx_second_entity>)

input
<radius>   *:= fillet radius*
<idx_first_shape> := index of the first shape
<idx_second_shape> := index of the second shape

note Given 2 shapes, join them with the desired radius. The versus of one of the 2 shapes may be changed.

example
```
  ~G0(100, 300, 5, 101)
  ~LET [IDX_1] = ~G1(300, 100, 5)
  ~LET [IDX_2] = ~G1(500, 300, 5)
  ~Fillet(100,  [IDX_2],  [IDX_1])
```

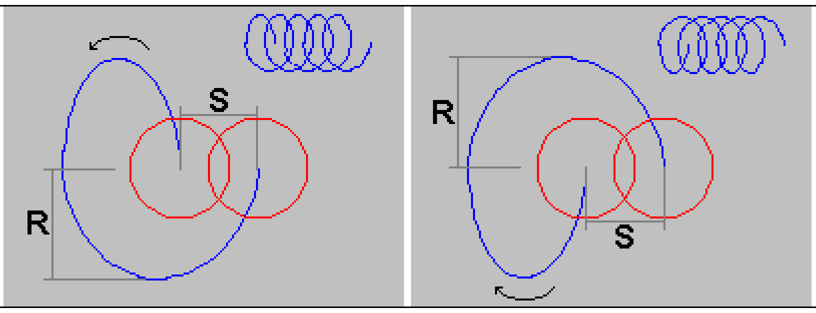
command: **~Epicycloid**
Make a polishing path starting from a source path

applied to:	post-processor	macro	drw rebuild
	[ ]	[X]	[ ]

See also: ~EpicycloidDlg

syntax
~LET [idx] = ~Epicycloid(<isA>, <idx>[, <param>])

input
<isA>,<idx> reference to the source path; it may be changed by this function,
see note below
<param> has four different syntaxes:
"MODEL=<NAME>" or
"Versus=<VERSUS>, Type=<TYPE>, RADIUS=<R>, STEP=<S>, STEPS/TURN=<ST>, DIAM=<diam>"   or
"Versus=<VERSUS>, Type=<TYPE>, RADIUS=<R>, STEP=<S>, STEPS/TURN=<ST>, T=<TOOLNUMBER>" or
"Versus=<VERSUS>, Type=<TYPE>, RADIUS=<R>, STEP=<S>, STEPS/TURN=<ST>, TC=<TOOLCODE>"
If <param> is not specified, the default model will be used.
Meaning for the parameters:

	
<NAME>	*The name of the model containing all the parameters*
<VERSUS>	*Rotation versus: 1 (c.clockwise) or 2 (clockwise)*
<TYPE>	*Path type: 0 (make mostly segments) or 1 (make mostly arcs). Type=0 makes segments but may create arcs near non-tangent points; Type=1 makes arcs but may create segments near non-tangent points*
<R>	*Epicycloid radius*

<S>	*Moving step for each turn*
<ST>	*Number of computed points for each turn (approximation)*
<diam>	*Tool diameter*
<TOOLNUMBER>	*Tool number*
<TOOLCODE>	*Tool code*
	T and TC exclude one each other. *Specifying MODEL the other parameters are ignored.* *The missing parameters are collected from the default model.* *You can Add/Delete/Edit models from inside the CAD:* *Draw→Objects→Polishing: select a path and press the "Edit/View details" button*

return The index of the first shape created or –1 if the epicycloid path can not be generated.

note

In case of any error in the input parameters, the macro stops running. If for some reason the path can not be executed, the function returns –1.
The source path is subject to routing reduction before applying the epicycloid, thus the shapes index <idx> may change, use idx2hnd() before calling this function if you need subsequent access to the source path.

example

```
  ~LET [idx_epi] = ~Epicycloid([isA],[idx],"MODEL=PREDEF")
  ~LET [idx_epi] = ~Epicycloid([isA],[idx]," STEP=40")
  ~LET [idx_epi] = ~Epicycloid([isA],[idx],"Versus=0, Type=1, RADIUS=90, STEP=40, STEPS/TURN=60, DIAM=-1,
TC=T1c")
```

command: ~EpicycloidDlg

Pops Up the Epicycloid Model dialog editor

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

See also: ~Epicycloid

syntax

~LET [CurrentModel] = ~EpicycloidDlg()

return

The name of the current model if Ok is pressed or an empty string if Cancel has been pressed.

note

This command allows the user to select/create/delete/edit a model as long as all the EpiCycloid parameters.

example

```
  ~LET [model] = ~EpicycloidDlg()
  ~IF [model] != [CHR0]
    ~LET [idx_epi] = ~Epicycloid([isA],[idx])
  ~ENDIF
```

command: ~ObjsMinMax

Computes the overall dimensions of the selected entities

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [qt] = ~ObjsMinMax("minx","maxx","miny","maxy","minz","maxz")

input

"minx","maxx","miny","maxy","minz","maxz" := the variables names for the result.
None of the parameters are compulsory, just leave blank the field if you don't need it.

return *The number of computed entites.*

note

One or more entities must be selected. If a shape inserted in a path is selected, the full path will be computed.

example

```
  ~LET [SEL_QT] = ~SELSINGLE("Select the entity")
  ~LET [qt] = ~ObjsMinMax("minx","maxx","miny","maxy")
  ~G0([minx],[miny],1,10)
  ~G1([maxx],[miny])
  ~G1([maxx],[maxy])
  ~G1([minx],[maxy])
```

```
~G1([minx],[miny])
```

command: ~InvertDir

invert the direction of a routing

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~InvertDir(<idx_entity>)

input

<idx_entity > := index of the first shape of the path to be inverted

note

Given an index of a shape, invert the path that begin with it. If the shape is assigned ah has some CAM data (as tool compensation) also them are correctly inverted

example

```
~SELGETOBJ(0,"isA","idx")
~InvertDir([idx])
~UNSELOBJ([isA],[idx])
```

command: ~ShapesReduction

Reduces the number of routings, if possible

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [idx] = ~ShapesReduction(<isA>, <idx>[, <params>])

input

<isA>, <idx > := index of any shape along the path
<params> := list of the reduction parameters.
If not specified the default parameters will be used.

return The index of the first shape of the reduced path. –1 in case of errors.

example

```
~SELGETOBJ(0,"isA","idx")
~UNSELOBJ([isA],[idx])
~LET [idx] = ~ShapesReduction([isA],[idx])
```

command: ~ParallelPocket

Creates an emptying path.

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax

~LET [idx] = ~ParallelPocket ("SEL", [<params>], ["retarray()"])

input

"SEL" := Base path(s) are all the currently selected entities.
<params> := list of pocketing parameters.
If not specified the default CAD parameters will be used.
"retarray()" := the name of the array where the index of the created will be put.
This parameter is optional.

Syntax for <params>: "<name>=<value>; …". Parameters are evaluated from left to right. Missing parameters will be defaulted to the latest used in Cad. You are suggested to always specify fT and rT or fTC and rTC. If the finishing Tool is not Tapered, sharp edges parameters will be ignored. Parameters are Case insensitive.

Name	**Description**	**example**
fT	Tool Number for finishing. NOTOOL stands for no finishing. Specify fT or fTC only.	fT=102
rT	Tool Number for roughing. NOTOOL stands for no roughing. Specify rT or rTC only.	rT=113
fTC	Tool Code for finishing. Specify fT or fTC only.	fTC=1270COMP
rTC	Tool Code for roughing. Specify rT or rTC only.	rTC=0318DOWN
Depth	Depth for the pocketing.	Depth=5.0
Overlap	Overlap in millimiters. Must be between 0 and the roughing tool diameter. Tapered Tools may have problems overlapping.	Overlap=1.0
Overlap%	Overlap as a percentage of the roughing tool diameter	Overlap%=10

	(alternate syntax for Overlap). Enter a value between 0 and 99.	
Angle	Path direction (in degrees) for pocket. Set 0 for Horizontal, 90 for Vertical, any other value between 0 and 90 for oblique.	Angle=45
SharpEdgesMaxAng	A positive value between 0 and 180 enables the Sharp Edges option. A Tapered finishing Tool must have been specified. You must explicitly write this parameter if you want Sharp Edges.	SharpEdgesMaxAng=179
Rdistance		
LeadInOut		

return -1 in case of error, a different value if Ok.

example
```
 ~SELGETOBJ(0,"isA","idx")
 ~LET [pparam] = "fT=NOTOOL; rT=103; Depth=5; Overlap=2; Angle=0"
 ~LET [ret] = ~ParallelPocket("SEL", [pparam],"resultidx()")
 ~UNSELOBJ([isA],[idx])
 ~IF [ret] = -1
   ~MSGBOX(,,"Error, Pocket NOT generated")
 ~ELSE
   ~boundsAry("resultidx",,,"qt")
   ~MSGBOX(,,"Pocket generated, ",[qt]," different paths have been created")
 ~ENDIF
```

macro LAYER commands

command: ~SetLayer

Add an entity to a specific layer.

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax
~LET [idx] = ~SetLayer(<isA>, <idx>, "nome_layer")

input
<isA>, <idx > := index of any shape along the path
“nome_layer” := name of the layer that you want the entity to be assigned to

return -1 in case of error, a different value if Ok.

example
```
 ~SELGETOBJ(0,"isA","idx")
 ~SetLayer([isA],[idx], default)
 ~UNSELOBJ([isA],[idx])
```

command: ~SetLayerProp

Assign to an entity the properties of the layer where it’s drawn.

applied to:	post-processor	macro	drw rebuild
	☐	X	☐

syntax
~LET [idx] = ~SetLayerProp(<isA>, <idx>, "diam_da_layer", "color_da_layer", "prof_da_layer", "profend_da_layer")

Note
"diam_da_layer", "color_da_layer", "prof_da_layer", "profend_da_layer"   = 1 --> TRUE
= 0 --> FALSE

input

<isA>, <idx >	*:= index of any shape along the path*
“diam_da_layer”	*:= assign to the entity the diameter value specified in the layer properties;*
“color_da_layer”	*:= assign to the entity the color specified in the layer properties;*
“prof_da_layer”	*:= assign to the entity the depth value specified in the layer properties;*
“profend_da_layer”	*:= assign to the entity the end depth value specified in the layer properties (unused for holes);*

return -1 in case of error, a different value if Ok.

example
```
~SELGETOBJ(0,"isA","idx")
~SetLayerProp([isA],[idx], 1,1,0,0)
~UNSELOBJ([isA],[idx])
```

command: **~SetColor**

Set the entity color using an RGB value (number included between 0 and 255).

applied to:	post-processor	macro	drw rebuild
	☐	☒	☐

syntax

~LET [idx] = ~SetLayerProp(<isA>, <idx>, "red", "green", "blue")

input

<isA>, <idx >	*:= index of any shape along the path*
“red”	*:= red component value of the RGB color (0-255);*
“green”	*:= green component value of the RGB color (0-255);*
“blue”	*:= blue component value of the RGB color (0-255);*

example
```
~SELGETOBJ(0,"isA","idx")
~SetColor([isA],[idx], 255,0,0)
~UNSELOBJ([isA],[idx])
```

macro NEW COMMANDS

command: **~GETOSNAP**

Allows to capture a position in the current face using the osnap

syntax

~OSNAP(<Type>, <isA>, <idx>, <Value X>, <Value Y>, <Value Z>)

input

<Type> *:= osnap mode to use:*
- *“E” for start/end point*
- *“C” for centre point*
- *“M” for middle point*
- *“N” for nearest point*
- *“I” for intersection point*
- *“T” for tangent point*

<isA>, <idx >	*:= index of any shape/hole*
“Value X”	*:= coordinate X of the point;*
“Value Y”	*:= coordinate Y of the point;*
“Value Z”	*:= coordinate Z of the point;*

example

~OSNAP([type], [idx], [x], [y], [z])

command: **~SetPathDir**

Allows to set the direction of the entity

syntax

~SetPathDir(<Dir>, <isA>, <idx>)

input

<Dir> := direction type: CwRightBottom or CcwLeftTop
<isA> := entity type, must be IS_SHAPE
<idx> := entity index

returns:
0 : the path is already Ok (doesn't need to be reversed)
1 : the path has been reversed

example
~SetPathDir(CwRightBottom, [isA], [idx])
NOTE:
the full path direction is set.

command: ~GetPathDir
Allows to get the direction of the entity

syntax
~GetPathDir(<isA>, <idx>)

input
<isA> := entity type, must be IS_SHAPE
<idx> := entity index

returns:
CwRightBottom
CcwLeftTop

example
~LET [dir] = ~GetPathDir([isA], [idx])
NOTE: the full path direction is considered.

command: ~MakeMacro
Allows to create a new macro command starting from the status of the current drawing

syntax
~MakeMacro(<MacroFileName>, <MacroBmpFileName>)

input
<MacroName>   := macro name
<MacroBmpFileName> := macro bitmap file name

example
~MakeMacro("MyNewMacro", "c:\Aspan4\macro\MyMacro\MyNewMacro.bmp")

command: ~SelectParametricType
Allows to select the parametric model

syntax
~SelectParametricType(<Parametric Type>)

input
<Parametric Type>   := the current parametric model type

returns:
the new parametric model type

example
~SelectParametricType(0)

command: ~ApplyParametricType
Allows to apply the selected parametric model to the current drawing

syntax
~ApplyParametricType(<Parametric Type>)

input
<Parametric Type>   := the current parametric model type

example

~ApplyParametricType(0)

command: ~GetParametricTypeStr

Allows to retrive the name of the selected parametric model

syntax

~GetParametricTypeStr(<Parametric Type>)

input

<Parametric Type>   := the current parametric model type

returns:

the name of the selected parametric model type

example

~LET [STRING] = ~GetParametricTypeStr(2)

command: ~SELSINGLEOBJ

Allows to select a single shape of a path

syntax

~SelSingleObj(<message>)

input

<message>   := string to show to the user

returns:

the name of the selected parametric model type

example

~LET [Idx] = ~SelSingleObj("Select a line or an arc of the path")

command: ~RotateDRW

Rotate the current drawing in X, Y and Z

syntax

~RotateDRW([<drwName>], <angle X>, <angle Y>, <angle Z>)

input

<drwName>	*:= name of the drawing to rotate*
<angle X>	*:= rotation angle in X*
<angle Y>	*:= rotation angle in Y*
<angle Z>	*:= rotation angle in Z*

example

~LET [RES] = ~RotateDRW(, 90, 0, 0)

command: ~Digitizer

Allow to create a spline starting from array of coordinates

syntax

~LET [idx] = ~Digitizer(<X values array>, <Y values array>, <Z values array>)

input

<X values array> := array of the X coordinates of the points
<Y values array> := array of the Y coordinates of the points
<Z values array> := array of the Z coordinates of the points

the arrays must have the same elements quantity and the same start index *command:* **~G03D**

Starting linear movement in the 3D space
Warning: preliminary specifications. Subject to modification.
Always test programs containing these instructions in simulation mode and pay attention: collisions may occur in the machine

syntax

~G03D(<x3d>, <y3d>, <z3d>, <depth>, <diam>, <tool XY inclination angle>, <tool tilting angle>, <formulas>)

input

<x3d>	*:= X coordinate in the space*
<y3d>	*:= Y coordinate in the space*
<z3d>	*:= Z coordinate in the space*
<depth>	*:= shape start depth*
<diam>	*:= tool diameter*
<tool XY inclination angle>	*:= tool vector angle (in XY plane) (0 ... 360) (See Note 1)*
<tool tilting angle>	*:= tool tilting angle (0 ... 180) (See Note 1)*
<formulas>	*:= tool lead-in side*

Coordinates explanation:
<x3d> = <y3d> = <z3d> = 0 -> means on the top of the top left corner
<x3d> positive to the right
<y3d> positive to the bottom
<z3d> positive to the machine plane

Same coordinates as for ~G0 and ~G1 commands
Angles explanation:
<tool tilting angle> = 0 : tool is vertical (perpendicular to the machine plane as for normal G0 and G1 instructions)
90 : tool is horizontal (parallel to machine plane)

<tool XY inclination angle> = supposing that <tool tilting angle> = 90
= 0 : tool points on the right O====>
= 90 : vector rotates 90 degrees clockwise O

= 180 : vector rotates 180 degrees clockwise <====O

NOTE:
The angles extension depends on the specific machine, some machines may not be able to execute some 3D workings

command: **~G13D**

Make a linear movement in the 3D space
Warning: preliminary specifications. Subject to modification.
Always test programs containing these instructions in simulation mode and pay attention: collisions may occur in the machine

See ~G03D for explanation of the parameters

syntax

~G13D(<x3d>, <y3d>, <z3d>, <depth>, <tool XY inclination angle>, <tool tilting angle>, <formulas>, <shape subtype>)

input

<x3d>	*:= X coordinate in the space*
<y3d>	*:= Y coordinate in the space*
<z3d>	*:= Z coordinate in the space*
<depth>	*:= shape end depth*
<tool XY inclination angle>	*:= tool vector angle (in XY plane) (0 ... 360)*
<tool tilting angle>	*:= tool tilting angle (0 ... 180)*

<shape subtype>   := shape subtype

command: **~SetPriority**

Allow to set the priority of the specified entity

syntax

~SetPriority([isA], [idx], <value>)

input

<value> := priority value

command: **~Ellipse**

Allow to create an ellipse specifying a center a two radius

syntax

~Ellipse(<xc>, <yc>, <radius_x>, <radius_y>, <Dir>, <diameter>, <depth>, <qt_arcs>)

input

<xc>, <yc> := coordinates of the center of the ellipse
<radius_x>, <radius_y> := radius in X and Y of the ellipse
<Dir> := direction type: CwRightBottom or CcwLeftTop
<qt_arcs> := number of arcs that compone the ellipse

example

~Ellipse(200, 200, 100, 50, “CwRightBottom”, 10, 10, 16)

command: **~SetDepth**

Allow to set the depth of the specified entity

syntax

~SetDepth(<isA>, <idx>, <depth>)

input

<isA>   = type of entities (IS_HOLE or IS_SHAPE)
<idx>   = object's index
<depth> = depth value

command: **~SetDiameter**

Allow to set the diameter of the specified entity

syntax

~SetDDiameter(<isA>, <idx>, <diameter>)

input

<isA>   = type of entities (IS_HOLE or IS_SHAPE)
<idx>   = object's index
<diameter> = diameter value

command: **~CreateLayer**

Allow to create a new layer

syntax

~ CreateLayer(“layer name”, “layer parameters string”)

input

“layer parameters string” = “varname_1 = varvalue_1, varname_2 = varvalue_2, ….”
Possible “varname”:
- UNUSABLE
- VISIBLE
- BLOCKED
- PRINTABLE
- START_DEPTH

- END_DEPTH
- DIAMETER
-ASSIGNABLE
- WORKABLE
- USECAMDATA
- TOOL or TOOL_CODE
- START_SPEED
- SPEED
- RPM
- PRIORITY
- COLOR_RED
- COLOR_GREEN
- COLOR_BLUE

command: ~ModifyLayer

Allow to modified an existing layer

syntax
~ ModifyLayer("layer name", "layer parameters string")

input
"layer parameters string" = "varname_1 = varvalue_1, varname_2 = varvalue_2, …."
Possible "varname":
- UNUSABLE
- VISIBLE
- BLOCKED
- PRINTABLE
- START_DEPTH
- END_DEPTH
- DIAMETER
-ASSIGNABLE
- WORKABLE
- USECAMDATA
- TOOL or TOOL_CODE
- START_SPEED
- SPEED
- RPM
- PRIORITY
- COLOR_RED
- COLOR_GREEN
- COLOR_BLUE

command: ~ExistLayer

Check the existence of the specified layer name

syntax
~ ExistLayer("layer name")

return 1 if the layer exist

command: ~Scale

Allow to scale the selected entity

syntax
~LET [idx] = ~Scale("isA", "idx", "value X", "value Y", "value Z", "Tcp X", "Tcp Y")

input
<isA> := type of entities (IS_HOLE or IS_SHAPE)
<idx> := object's index
<value X> := scale X value
<value Y> := scale Y value
<value Z> := scale Z value
<TCP X> := scale origin X coordinate
<TCP Y> := scale origin Y coordinate

Appendix: A

Specifying ~MSG(<message id>), the message will automatically be translated in your language.

file_not_open	file_open_error	file_close_error
file_read_error	file_write_error	file_seek_error
file_dir_not_sel	file_dir_not_found	file_no_filename
file_max_dir	file_not_free	file_bad_file
file_bad_version	file_bad_size	file_no_mem
file_not_found	file_unknown_error	info_msg
warning_msg	error_msg	fatal_msg

example

~MSG(warning_msg)

Appendix: B

List of the math and boolean operators that can be evaluated:

name	alt.name	Description	example
"**"	"^"	POWER ELEVATION	~(10 ** 2)
"-"		SUBTRACTION	~(10 - 5)
"-"		unary -	~(-[A])
"+"		ADDITION	~(3 + 4)
"*"		MULTIPLICATION	~(4 * 5)
"/"		DIVISION	~(7 / 3)
"="	"=="	EQUAL COMPARISON	~I([A] == [B])
">="	"=>"	GREATER or EQUAL	~I([A] => [B])
">"		GREATER	~I([A] > [B])
"<>"	"!="	DIFFERENT	~I([A] <> [B])
"<="	"=<"	LESS or EQUAL	~I([A] <= [B])
"<"		LESS	~I([A] < [B])
"("		OPEN parentesis	~(([A] + [B]) * [C])
")"		CLOSE parentesis	
"MOD"	"%"	REMAINDER	~I([A] MOD [B])
"PI"		PI-greek 3.14	
"DX"		variable: panel X size	
"DY"		variable: panel Y size	
"DZ"		variable: panel Z size	
"FLD"		variable: machine working field	
"ABS"		ABSOLUTE VALUE	~(ABS([A]))
"ACOS"		ACOS	~(ACOS(0.5))
"ASIN"		ASIN	~(ASIN(0.5))
"ATAN2"		ATAN2	~(100 ATAN2 50)
"ATAN"		ATAN	~(ATAN(3.47))
"COS"		COS	~(COS(45))
"SIN"		SIN	~(SIN(45))
"TAN"		TAN	~(TAN(45))
"RD"		Round Down	~(RD(12.8))
"RU"		Round Up	~(RU(12.8))
"EXP"		exponential e to the x	~(EXP([x]))
"LN"		the natural logarithm	~(LN([x]))
"LOG"		the base 10 logarithm	~(LOG([x]))
"SQR"	"SQRT"	the positive square root	~(SQRT([x]))
"OR"	"||"	logical OR	~(([A] < [B}) OR ([B] < [C]))
"AND"	"&&"	logical AND	~(([A] < [B}) AND ([B] < [C]))
"XOR"	"^^"	logical XOR	~(([A] < [B}) XOR ([B] < [C]))
"NOT"	"!"	logical NOT	~(!([A] < [B}))
"BOR"	"|"	binary OR	~I([A] | 16)
“BAND"	"&"	binary AND	~I([A] & 16)
"BXOR"	"^"	binary XOR	~I([A] ^ 16)
"BNOT"	"~"	binary NOT	~I(~[A])
"SHL"	"<<"	binary shift left	~I(1 << 4)

"SHR"	">>"	binary shift right	~I(16 >> 4)

Appendix: C

List of the common (special) variables:
[CHR91] Open square bracket, character "[", same as ~CHR(91)
[CHR93] Close square bracket, character "]", same as ~CHR(93)
[TILDE] character "~", same as ~CHR(126)
[CHR13] New-Line control character, same as ~CHR(13)
[CHR0]   Null character (empty string): ""

Appendix: D

Where a file name has to be specified for read and/or write, you can specify a path or not.
It a path is not given, the currend directory will be used; if a path is given it will be used.

example
```
 ~LET [file1] = C:\Autoexec.bat
 ~LET [file2] = def.tlg
```

Appendix: E

Decoder variables

List of the Decoder variables and constants available when generating the Part-Program.

NAME	VALUE	NAME	VALUE
CHR0		CHR13	
CHR91	[	CHR93	]
TILDE	~	PROGDIR	C:\ASPAN4\
PART_DIR	C:\ASPAN4\P2PART\T95	LANGUAGE	ITALIANO
NULL	NULL	MAXPPLINES	0
CALLER	M	XSUB	0.00
SUBNAME	NULL	YSUB	0.00
PAN_DX	750.00	PAN_DY	300.00
PAN_DZ	20.00	CFG	T95L_NUM
ATR	DEF	REM1	
REM11	NULL	REM12	NULL
REM2	NULL	REM21	NULL
REM22	NULL	FACE_UPPER	FACE_UPPER
FACE_LEFT	FACE_LEFT	FACE_RIGHT	FACE_RIGHT
FACE_FRONT	FACE_FRONT	FACE_REAR	FACE_REAR
FACE_NOORTHO	FACE_NOORTHO	FACE	1
TOOL_CR	0	TOOL_CR_PROF	0
TOOL_CR_S	0.00	X	395.41
Y	146.48	Z	1.00
ZMAX	1.00	XC	
YC		CEN_RAD	0
RADIUS		ARC_ANGLE	
ENABLE_XBO	0	HOLE_DIAM	NULL
FACE_OPTIMIZE	0	E	NULL
FEED	4	RPM	2800
TTYPE_NONE	TTYPE_NONE	TTYPE_BIT	TTYPE_BIT
TTYPE_ROUTER	TTYPE_ROUTER	TTYPE_SAW	TTYPE_SAW
TTYPE_CONIC	TTYPE_CONIC	TTYPE_SENSOR	TTYPE_SENSOR
TTYPE_SPECIAL	TTYPE_SPECIAL	BITTYPE_NONE	BITTYPE_NONE
BITTYPE_BRAD	P	BITTYPE_SPEAR	L
BITTYPE_SUNK	S	TOOL	1
TTYPE	TTYPE_BIT	TOOL_CODE	
TOOL_DESCR		BITTYPE	P
SVAS_HEIGHT	0.00	T_LENGTH	30.00
T_LENGTH_TOT	64.30	T_LENGTH_MAX	64.30
T_RADIUS	5.00	ROT_DIR	CW
T_X	0.00	T_Y	0.00

T_Z	0.00	T_ANG	0.00
T_ZANG	0.00	T_PMAG	-1
VECTOR_ANG	NULL	TOOL_PREV	
TTYPE_PREV		H	
ANG		PLANE	
CNTYPE	10400	CNMODEL	1
USE_660	NO	SPEAR_BIT_PROTR	5.00
IS_A	IS_NOTHING	IS_A_NEXT	IS_NOTHING
FACE_NEXT	-1	HEAD_X	-32000
HEAD_Y	-32000	HEAD_Z	-32000
FACE_PREV	1	TOOL_CR_PREV	0
TOOL_CR_PROF_PREV	0	TOOL_CR_S_PREV	0.00
X_PREV		Y_PREV	
XC_PREV		YC_PREV	
Z_PREV		ZMAX_PREV	
H_PREV		E_PREV	0.00
RPM_PREV		FEED_PREV	
IS_A_PREV		PART_DATE	2003/01/23
PART_TIME	12:56:19	LASTINSTRISG0	1
APP	0	APP_PREV	NULL
FIRST_HOLE	1	FIRST_SHAPE	1
HOLE_DONE	0	SHAPE_DONE	0
NUM_BLOCK	1	DOUBLE_PANEL	0
MAIN_SPINDLE	42	TWIN_SPINDLE	48
PP_TOOL_CR	0	PP_TOOL_CR_PROF	0
PP_TOOL_CR_S	0	T_SPEED	1
MANUAL_TC	0	NOP_AT_END	0
ETOOL_PREV	0	MANUAL_TC_Y	0
MANUAL_TC_V	9	NOP_SXDX	0
NOP_FROREA	0	NOP_USER_X	0
NOP_USER_Y	0	NOP_USER_V	9
VTENDVT	1	V_S_FROM_MAC	0
FACE_LOWER	0	WRITE_GIN	0
LEAD_TYPE	LEAD_NONE	IS_LEAD_IN	1
FORCE_PLANE_CHANGED	0	xSUB_CUT_ID	;--S--
xSUB_CUT_CMD	;--S-CMD=	PLANE_ORI_X	0.00
PLANE_ORI_X_PREV		PLANE_ORI_Y	0.00
PLANE_ORI_Y_PREV		PLANE_ORI_Z	0.00
PLANE_ORI_Z_PREV		PLANE_ROT_Z	0.00
PLANE_ROT_Z_PREV		PLANE_ROT_X	0.00
PLANE_ROT_X_PREV		PLANE_TOOLROT_Z	0.00
PLANE_TOOLROT_Z_PREV		PLANE_TOOLROT_X	0.00
PLANE_TOOLROT_X_PREV		PLANE_CHANGED	0
FACE_PLANE	33	EXIST_FX	0
EXIST_FY	0	ISINCH	0
RIP	1	FIELD	A
UDM	MM	CONT	0
PGM_USE_ORG	0	PGM_ORG_X	0.00
PGM_ORG_Y	0.00	PGM_ORG_Z	0.00
PART_NAME	PIO	USE_BORDA	NO
BORDA_FEED_INI		BORDA_FEED	
A		D	
B		B_CLOSE	
R		I	
J		L	
AUTOSTART		EIN_CUT	
TRulli		ROT	
V		Q	
TNAV		C_COINI	
D_SBINI		C_SBINI	
D_R1INI		C_R1INI	
D_R1CI		C_R1CI	
D_R2INI		C_R2INI	
XMIN		YMIN	
AMIN		RMIN	
S		BMIN	
VOUT		BROT	
TCSPI		AROT	
QROT		QLAMP	
DLAMP		XCUT	
YCUT		ACUT	

BCUT		DCUT	
SCUT		QCUT	
GCUT		TOOLCUT	
CORRUT		LADD	
TOOLBORDA		ANGREFRASCH	
IC_VALUE		MACHINE_TYPE	0
NAV_ESCLUSION		TIN	
LAMPON_G0		V_CAR	
ALT_BORDO		SPES_BORDO	
BORDA_FEED_PREV		MULTIBORDA	0
R_ANG	0.00	WORKINGTYPE	0
A_PREV	0.00	D_PREV	0.00
B_PREV	0.00	R_PREV	0.00
I_PREV	0.00	J_PREV	0.00
L_PREV	0.00	AUTOSTART_PREV	0
EIN_CUT_PREV	0	TRulli_PREV	0
ROT_PREV	0	ALT_BORDO_PREV	0.00
SPES_BORDO_PREV	0.00	V_PREV	0.00
Q_PREV	0.00	TNAV_PREV	0
C_COINI_PREV	0.00	D_SBINI_PREV	0.00
C_SBINI_PREV	0.00	D_R1INI_PREV	0.00
D_R1INI_PREV	0.00	C_R1INI_PREV	0.00
D_R1CI_PREV	0.00	C_R1CI_PREV	0.00
D_R2INI_PREV	0.00	C_R2INI_PREV	0.00
NAV_ESCLUSION_PREV	0	TIN_PREV	0.00
LAMPON_G0_PREV	0	V_CAR_PREV	0.00
MULTIBORDA_PREV	0	B_CLOSE_PREV	0.00
R_ANG_PREV	0	XMIN_PREV	0.00
YMIN_PREV	0.00	AMIN_PREV	0.00
RMIN_PREV	0.00	S_PREV	0.00
BMIN_PREV	0.00	VOUT_PREV	0.00
BROT_PREV	0.00	TCSPI_PREV	0.00
QT_LAV	1	WORKCOUNT	0
H_NOT_WRITTEN	0	SeqF6	0
QtCadIndexes	number of indexes of CAD entities involved in machining	CadIndexes	Array of indexes of Cad entiies involved in machining
AssignedToolsArray	array of tool used to process the entities	QtAssignedToolsArrayItems	number of items in “AssignedToolsArray”
ShapeCadIdx	cad index of the current processed shape	FirstHoleCadIdx	cad index of the current processed hole (or the first of a drop)

Appendix: F

Part-Program POST-PROCESSOR skeleton

Dialog window for data input proposed when generating a part-program.
Variables here defined (**PROGRAMMATORE** and **NUM_JOB**) may be used everywhere during program execution.
WARNING : two variables are automatically inserted by the procedure:
part-program Directory and **program name**

```
~EXTRA (ITALIANO)
[PROGRAMMATORE], "Nome del programmatore",C,Cristiano
[NUM_JOB],      "Numero commessa",I,105
~END

~EXTRA (ENGLISH)
[PROGRAMMATORE], "Programmer's name",C,Cristiano
[NUM_JOB],      "Job #",I,105
~END

-------------------------------------------------------
         Variables Initialization
 This is the first Post-Processor function called.
-------------------------------------------------------
~START INIT
  ~?BEGIN INIT FUNCTION
  ~?  "Processor" initialization,
                   Write here the initialization of the variables used in the post-processor and the codes of machine initialization
                   that does not depend from panel dimension, to be written in the constructor (HEADER)
                   example specify to Aspan that the tool numbers are not grouped together
  ~LET [QT_GROUP] = 0
  #
  ~? END INIT FUNCTION
~END

#
------------------------------------------------------------
   START-UP AND PART-PROGRAM INITIALIZATION
------------------------------------------------------------
~START HEADER
  ~? BEGIN HEADER FUNCTION
  ~?  Program initialization,
                   Here are available the variables of the program we are processing
  ~?  Language set: [LANGUAGE]
  ~?  Program name: [PART_NAME] (folder : [PGM_DIR])
  ~?  Program created on [PART_DATE] at time [PART_TIME]
  ~?  Panel dimension X=[PAN_DX] Y =[PAN_DY] Z=[PAN_DZ]
  ~? END HEADER FUNCTION
~END

--------------------------------------------------------
                 REMARKS
--------------------------------------------------------
~START REMARKS
..~? BEGIN REMARKS FUNCTION
                   It is possible to transfer in the generating program the first two rows of the remark associated to the drawing,
                   written by the user
    ~IF [REM11] = [NULL]
      ~?( [REM1] )
    ~ELSE
      ~?( [REM11][REM12])
    ~ENDIF
    ~IF [REM2] != [NULL]
      ~IF [REM21] = [NULL]
        ~?( [REM2] )
      ~ELSE
        ~?( [REM21][REM22] )
      ~ENDIF
    ~ENDIF
..~? END REMARKS FUNCTION
~END

---------------------------------------------------------
                 ORG_MAC
---------------------------------------------------------
~START ORG_MAC
..~? BEGIN ORG_MAC FUNCTION
                   Used to setup the part-program origin
..~? END ORG_MAC FUNCTION
~END

---------------------------------------------------------
                      HOLE
---------------------------------------------------------
~START HOLE
  ~? BEGIN HOLE FUNCTION
  ~? BORING:
  ~?   position X=[X] Y=[Y] depth Z=[Z]
  ~?   Tool/s [T] Boring feed = [FEED] m/min
  ~? END HOLE FUNCTION
~END

---------------------------------------------------------
                      ANG_HOLE
---------------------------------------------------------
~START ANG_HOLE ~MODAL(X=,Y=,Z=,V)
~END

---------------------------------------------------------
                      NOP
---------------------------------------------------------
```

```
~START NOP ~MODAL(X=,Y=,V)
~END

-----------------------------------------------------------
                    FACE
-----------------------------------------------------------
~START FACE
  ~IF [FACE] != [FACE_PREV]
      ~? BEGIN FACE FUNCTION
      ~? Set working face
      ~?   Previous working face [FACE_PREV]
      ~IF [FACE] = 1
              ~?   Working face to be set: UPPER
      ~ENDIF
      ~IF [FACE] = 2
              ~?   Working face to be set: LEFT
      ~ENDIF
      ~IF [FACE] = 3
              ~?   Working face to be set: RIGHT
      ~ENDIF
      ~IF [FACE] = 4
              ~?   Working face to be set: FRONT
      ~ENDIF
      ~IF [FACE] = 5
              ~?   Working face to be set: BACK
      ~ENDIF
      ~IF [FACE] = 6
              ~?   Working face to be set: LOWER
      ~ENDIF
      ~? END FACE FUNCTION
  ~ENDIF
~END

-----------------------------------------------------------
          TOOL_CORR (called from PPgen)
-----------------------------------------------------------
~START TOOL_CORR
~END

-----------------------------------------------------------
                    TAST
-----------------------------------------------------------
~START TAST ~MODAL(X=,Y=)
~END

-----------------------------------------------------------
             WRITE LEAD IN/OUT ISTRUCTIONS
-----------------------------------------------------------
~START WRITE_GIN_GOUT
~END

-----------------------------------------------------------
                 INIT_SHAPE(G0)
-----------------------------------------------------------
~START INIT_SHAPE
  ~? BEGIN INIT_SHAPE FUNCTION
    ~?Routing start machining, tool attivation [T], put the machine
    ~?to position [X],[Y] at quick feed,
    ~?then drop at feed [FEED]m/min to working depth [Z]
  ~? G0 X=[X] Y=[Y] TOOL=[T]
  ~? G1 Z=[Z] F[FEED]
  ~? END INIT_SHAPE FUNCTION
~END

----------------------------------------------------------
                 END_SHAPE(G0)
----------------------------------------------------------
~START END_SHAPE
  ~? BEGIN END_SHAPE FUNCTION
                 "Close" the routing machining moving the machine head out of material at safety distance
  ~? END END_SHAPE FUNCTION
~END

-----------------------------------------------------------
```

```
                  LIN_SHAPE(G1)
-----------------------------------------------------------
~START LIN_SHAPE ~MODAL(Z,F)
  ~?G1 X[X] Y[Y] Z[Z] F[FEED] (LIN_SHAPE FUNCTION)
~END

-----------------------------------------------------------
                  G2_CIRC_SHAPE(G2)
-----------------------------------------------------------
~START G2_CIRC_SHAPE ~MODAL(Z,F)
  ~?G2 X[X] Y[Y] Z[Z] I[I] J[J] F[FEED] (G2_CIRC_SHAPE FUNCTION)
~END

-----------------------------------------------------------
                  G3_CIRC_SHAPE(G3)
-----------------------------------------------------------
~START G3_CIRC_SHAPE ~MODAL(Z,F)
  ~?G3 X[X] Y[Y] Z[Z] R=[RADIUS] F[FEED] (G3_CIRC_SHAPE FUNCTION)
~END

-----------------------------------------------------------
                  TG_SHAPE(G5)
-----------------------------------------------------------
~START TG_SHAPE ~MODAL(X=,Y=,Z=,V)
~END

-----------------------------------------------------------
                  INIT_ANGSHAPE(G0R)
-----------------------------------------------------------
~START INIT_ANGSHAPE ~MODAL(X=,Y=,Z=,H=,V)
~END

-----------------------------------------------------------
                LIN_ANGSHAPE(G1R)
-----------------------------------------------------------
~START LIN_ANGSHAPE ~MODAL(X=,Y=,Z=,H=,V)
~END

-----------------------------------------------------------
                G2R_CIRC_SHAPE(G2R)
-----------------------------------------------------------
~START G2R_CIRC_SHAPE ~MODAL(X=,Y=,Z=,H=,V)
~END

-----------------------------------------------------------
                G2R_CIRC_SHAPE(G2R)
-----------------------------------------------------------
~START G3R_CIRC_SHAPE ~MODAL(X=,Y=,Z=,H=,V)
~END

-----------------------------------------------------------
          CALLED BEFORE MAKING HOLES
-----------------------------------------------------------
~START BEFORE_HOLES
~END

-----------------------------------------------------------
         CALLED AFTER MAKING HOLES
-----------------------------------------------------------
~START AFTER_HOLES
~END

-----------------------------------------------------------
          CALLED BEFORE MAKING ROUTINGS
-----------------------------------------------------------
~START BEFORE_SHAPES
~END

-----------------------------------------------------------
          CALLED AFTER MAKING ROUTINGS
-----------------------------------------------------------
~START AFTER_SHAPES
~END
```

```
-----------------------------------------------------------
         CALLED AFTER MAKING EVERYTHING
-----------------------------------------------------------
~START END_PARTPROG
  ~? BEGIN END_PARTPROG FUNCTION
                Called to "close" the part-program, it may be used to position the machine in a "end of machining" position
                and to insert codes for motors switching off, vacuum deactivation, etc
  ~? END END_PARTPROG FUNCTION
~END

==========================================================
------------------------------------------------------------
           WRITE THE VACUUM CUPS TABLE
------------------------------------------------------------
~START SET_PLANE_POS
~END
```

Appendix: G

Object Properties

if [isA] = IS_HOLE:

".diam"	".dowprot"	".face"
".form"	".machining.proc"	".p"
".svas"	".x"	".y"
“.userdata”		

if [isA] = IS_OBJ_BB:

". xs "	". ys "	". xe "
". ye "	".p "	". diam"
".svas"	".dowprot"	". face"
“.form”	".machining.proc"	

if [isA] = IS_SHAPE:

".ange"	".angs"	".cut"
".diam"	".face"	".first"
".form"	".last"	".len"
".machining.proc"	".next"	".pbulge"
".pc"	".pe"	".prev"
".ps"	".radius"	".tange"
".tangs"	".xc"	".yc"
".xe"	".ye"	".xs"
".ys"	“.userdata”	

".tcomp":

0: no compensation	1: right compensation	2: left compensation	3: depth compensation

".type":

0: line	1: CounterClock-Wise Arc	2: Clock-Wise Arc

Appendix: H

CAD related variables

[DX]	panel <x> dimension
[DY]	panel <y> dimension
[DZ]	panel <depth>
[CURFACE]	current panel face
[_X]	current <x> quote
[_Y]	current <y> quote
[_Z]	current <depth> quote
[_XC]	current <xc> quote (arc center)
[_YC]	current <yc> quote (arc center)
[_ZC]	current <zc> quote (arc center)
[_R]	current <r> quote (arc radius)
[_DIAM]	current tool diameter

[TCOMP] current tool compensation
[SVAS]
[DOWPROT] current Dowel Protrusion
[SCUT] current Cut flag for shapes
[SCOLOR] Color flag
[OSNAPBAR] Current Osnap configuration; it is a bit-mapped integer variable.
Set the bit to enable the Osnap type:

useKBD	0x00001
osnap	0x00002
start	0x00004
chor	0x00008
tl1	0x00010
ta1	0x00020
tl2	0x00040
ta2	0x00080
tl3	0x00100
ta3	0x00200
pl1	0x00400
pa1	0x00800
pl2	0x01000
pa2	0x02000
pl3	0x04000
pa3	0x08000
cancel	0x10000
tgP	0x20000

Appendix: I

Tool diameter and Machining parameters

Instead of specifying the tool diameter, you can set the tool number and some other machine parameters separated by commas.

Property	***example***
T=<tool> or TC=<tool code>	T=101 or T=NOTOOL
R=<rpm>	R=3
SF=<Start Feed>	SF=2
F=<Feed>	F=8
PRY=<Machining priority> (lower number= higher priority), default value 500	PRY=400
PASS=<the number of machining passes>	PASS=2
STEP_DEPTH=<depth of each working step>	STEP_DEPTH=1
STEP_INVERTED	
INVERTED	
NOP	
TAST	
G5	
ZRAPIDEND	
ZWORKOUT	
D=<diameter>	D=10.0
DEPTH=<depth> [SEE NOTE1]	DEPTH=5
PROC=<PGM procedure>	PROC=HdeCHIP
LIN=<LeadIn> (read only)	
LOUT=<LeadOut> (read only)	

If specified T= or TC= and the tool does not exist or is invalid, the other parameters are ignored. Any invalid parameter is ignored.
You can specify multimachining data by filling an array: each element is for one machining. The maximum number of machinings is 4 (multimachining index must be between 0 and 3) for each hole and 4 (multimachining index must be between 0 and 3) for each shape.
NOTE1:
The depth of 1st pass MUST always be the same of the related CAD entity one.

Simple diameter example:
~G0(100, 300, 5, 10.0)

Tool data example:

```
~G0(100, 300, 5, ”T=101,SF=2,F=8”)
```

Multimachining example:

```
# Make a hole by a 2 step multimachining using 2 different tools
~DimAry("CAMD",0,4)
~LETARY("CAMD", 0) = "T=10, F=4000,PRY=500"
~LETARY("CAMD", 1) = "T=14, F=4800,PRY=501"
~B(140,200,5,"CAMD()")
~DELARY("CAMD")
```

Appendix: J

C++ development (internal use only)

Further commands can be easily added.
Look at Execute_UnknownFunc() in Common\Decoder.dll\Extdec.cpp
You have to recognize the Command name.

example

```
xdec = new ExpandedDecoder(w);
...
xdec->SetUserFunc(Dec_UserFunc);
```

Basic commands..1
command: ***~IF***..1
command: ***~FOR ... ~NEXT***..1
command: ***~DO ... ~LOOP***..2
command: ***~CALL***..2
command: ***~START***..3
command: ***~EXIT_SUB***..3
command: **#**..3
command: ***~LET***..3
command: ***~GOTO***..4
command: ***~:***..4
command: ***~DEBUG***..4
command: ***~MSGBOX***..4
command: ***~MSG***..5
command: ***~()***..5
command: ***~I()***..5
command: ***~EVAL***..5
command: ***~?, ~??***..6
command: ***~GetDecimalPrecision***..6
command: ***~SetDecimalPrecision***..6
command: ***~SetOutPutFile***..7
command: ***~FOPEN***..7
command: ***~FDIALOG, ~_FDIALOG***..8
command: ***~FolderSel***..8
command: ***~FOPENCOMM***..9
command: ***~FCLOSE***..9
command: ***~FREADB***..10
command: ***~FREADLINE***..10
command: ***~FWRITEB***..10
command: ***~FWRITE***..10
command: ***~FSEEK***..11
command: ***~FTELL***..11
command: ***~CMD***..11
command: ***~PRINT***..12
Expanded commands..12
command: ***~EXISTV***..12
command: ***~GETV ~_GETV***..12
command: ***~SETV***..12
command: ***~WRITEB***..12
command: ***~READB***..13
command: ***~READLINE***..13
command: ***~FEXIST***..14
command: ***~FSIZE***..14
command: ***~COPYFILE***..15
command: ***~DELFILE***..15
command: ***~MKDIRS***..15

command: ***~DELTREE*** *15*
command: ***~CWD*** *16*
command: ***~ChDir*** *16*
command: ***~FILEFIND*** *16*
command: ***~DIRFIND*** *17*
command: ***~CompletePath*** *17*
command: ***~Splitpath*** *17*
command: ***~ZIP*** *17*
command: ***~EXECUTE*** *18*
command: ***~PUTDEF*** *18*
command: ***~GETDEF, ~_GETDEF*** *18*
command: ***~LOADDEFS*** *19*
command: ***~SAVEDEFS*** *19*
command: ***~CLOSEDEFS*** *20*
command: ***~PUSHCHDEFS, ~_PUSHCHDEFS*** *20*
command: ***~POPDEFS ~_POPDEFS*** *20*
command: ***~MKDEFS, ~MKDEFS*** *21*
command: ***~CHDEFS, ~_CHDEFS*** *21*
command: ***~RMDEFS, ~_RMDEFS*** *21*
command: ***~IniFileRead, ~IniFileWrite*** *22*
command: ***~ShiftPoint*** *22*
command: ***~Get3DStepInfos*** *23*
command: ***~OBJGET*** *23*
command: ***~ExtractParams*** *24*
command: ***~ASC*** *24*
command: ***~CHR*** *24*
command: ***~STRCAT*** *25*
command: ***~MID*** *25*
command: ***~LEFT*** *25*
command: ***~RIGHT*** *25*
command: ***~INSTR*** *26*
command: ***~LEN*** *26*
command: ***~TRIM*** *26*
command: ***~CASE*** *26*
command: ***~Compare*** *27*
command: ***~SETSTATUS*** *27*
command: ***~dimAry ~boundsAry ~delAry ~Ary ~LETARY ~RENARY ~COPYARY*** ... *28*
command: ***~ary*** *29*
command: ***~Struct*** *29*
command: ***~Dims*** *29*
command: ***~Sname*** *30*
command: ***~CfgGet*** *30*
command: ***~CfgPut*** *31*
command: ***~CfgWrite*** *31*
command: ***~Atr*** *31*
command: ***~SetTool*** *32*
command: ***~GetTool*** *32*

Dialog commands 32
*command: **~EXTRA**..33*
*command: **~INPUT, ~_INPUT, ~INPUT_OK**..33*
*command: **~DlgBegin**..34*
*command: **~DlgItem**..34*
*command: **~DlgItemProperty**..35*
*command: **~DlgExecute**..36*
*command: **~DlgCommand**..37*
*command: **~DlgEnd**..37*
*command: **~TimeGet**..37*
*command: **~Time**..37*
*command: **~TimeElapsed**..38*
macro General commands 38
*command: **~g_LET**..38*
*command: **~RUN, ~RUNCMD**..38*
*command: **~G0, ~MOVE**..39*
*command: **~G1, ~LINE**..39*
*command: **~alG1, ~LINE_al**..40*
*command: **~G2, ~ARC_CW**..40*
*command: **~rG2, ~ARC_rCW**..40*
*command: **~G3, ~ARC_CCW**..41*
*command: **~rG3, ~ARC_rCCW**..41*
*command: **~ARC_3P**..42*
*command: **~G5, ~ARC_TG**..42*
*command: **~arG5, ~ARC_arTG**..43*
*command: **~B**..43*
*command: **~BB**..43*
*command: **~DIAM**..44*
*command: **~C**..44*
*command: **~F**..45*
*command: **~DIM**..45*
*command: **~SETPANEL**..46*
*command: **~Offset**..46*
*command: **~OffsetA**..46*
*command: **~TextSet**..47*
*command: **~Text**..48*
*command: **~CreateBmp**..48*
*command: **~MergeBmp**..48*
*command: **~EditLabel**..49*
*command: **~PrnLabels**..49*
*command: **~ObjsJoin**..49*
*command: **~ObjCut**..50*
*command: **~ObjDel**..50*
*command: **~Idx2Hnd**..50*
*command: **~Hnd2Idx**..50*
macro Selection commands 51
*command: **~GETPOINT**..51*

command: ***~GETSEGMENT*** *51*
command: ***~SELSINGLE*** *51*
command: ***~SELMULTI*** *51*
command: ***~SELOBJS*** *52*
command: ***~SELOBJ*** *52*
command: ***~UNSELOBJ*** *52*
command: ***~SELRECORD*** *53*
command: ***~SELGETOBJ*** *53*
command: ***~SELDELOBJ*** *53*
command: ***~SEL_QT*** *53*
command: ***~OBJSET*** *54*
command: ***~OBJS_QT*** *54*

macro CAM commands 54

command: ***~EDGEBANDING*** *54*
command: ***~EDGEBANDINGCHECKLENGHT*** *55*
command: ***~EDGEBANDINGCHECKLENGHT*** *55*
command: ***~GETTANGENTPATH*** *55*
command: ***~ENDTRIMMING*** *55*

macro Display commands 56

command: ***~ToUserCoordinates*** *56*
command: ***~FromUserCoordinates*** *56*
command: ***~UpdateWin*** *56*
command: ***~ZoomExt*** *57*
command: ***~PickTool*** *57*
command: ***~SetTool*** *57*
command: ***~GetTool*** *57*
command: ***~Fillet*** *58*
command: ***~Epicycloid*** *58*
command: ***~EpicycloidDlg*** *59*
command: ***~ObjsMinMax*** *59*
command: ***~InvertDir*** *60*
command: ***~ShapesReduction*** *60*
command: ***~ParallelPocket*** *60*

macro LAYER commands 61

command: ***~SetLayer*** *61*
command: ***~SetLayerProp*** *61*
command: ***~SetColor*** *62*

NEW COMMANDS

command: ***~GETOSNAP*** *61*
command: ***~SetPathDir*** *61*
command: ***~GetPathDir*** *61*
command: ***~MakeMacro*** *61*
command: ***~SelectParametricType*** *61*
command: ***~ApplyParametricType*** *61*
command: ***~GetParametricTypeStr*** *61*
command: ***~SelectHardware*** *61*

command: ~InsertHardware 61
command: ~SELSINGLEOBJ 61
command: ~GETEDGEBANDINGIDX 61
command: ~GetEdgeBand 61
command: ~RotateDRW 61
command: ~SplitDRW 61
command: ~RunCad 61
command: ~AutoAss 61
command: ~MakePP 61
command: ~CabinetPreview 61
command: ~SetLeadInOut 61
command: ~Digitizer 61
command: ~G03D 61
command: ~G13D 61
command: ~SetPriority 61
command: ~Ellipse 61
command: ~SetDepth 61
command: ~SetDiameter 61
command: ~Rotate 61
command: ~CreateLayer 61
command: ~ModifyLayer 61
command: ~ExistLayer 61
command: ~Scale 70

Appendix: A 68
Appendix: B 68
Appendix: C 69
Appendix: D 69
Appendix: E 69
Appendix: F 71
Appendix: G 75
Appendix: H 75
Appendix: I 76
Appendix: J 77