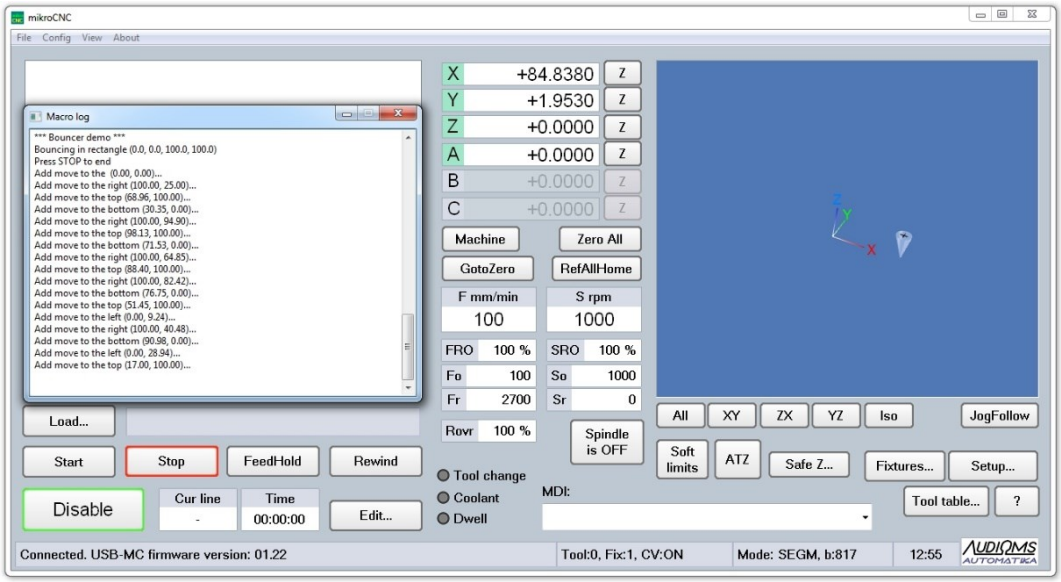
mikroCNC Scripting

User scripting support documentation



www.audiohms.com

TABLE OF CONTENTS

1 Python interpreter

The mikroCNC program that is used for controlling Audioms Automatika motion controllers has an embedded Python 3.x script interpreter. A large part of the syntax and functions of this programming language is supported.

Scripting has been supported since the software version mikroCNC V0.46.

Name	Example	Supported
If Else	`if..else..elif`	✓
Loop	`for/while/break/continue`	✓
Function	`def f(x,*args,y=1):`	✓
Subclass	`class A(B):`	✓
List	`[1, 2, 'a']`	✓
ListComp	`[i for i in range(5)]`	✓
Slice	`a[1:2], a[:2], a[1:]`	✓
Tuple	`(1, 2, 'a')`	✓
Dict	`{'a': 1, 'b': 2}`	✓
F-String	`f'value is {x}'`	✓
Unpacking	`a, b = 1, 2`	✓
Star Unpacking	`a, *b = [1, 2, 3]`	✓
Exception	`raise/try..catch..finally`	✓
Dynamic Code	`eval()/exec()`	✓
Reflection	`hasattr()/getattr()/setattr()`	✓
Import	`import/from..import`	✓
Context Block	`with <expr> as <id>:`	✓
Type Annotation	`def f(a:int, b:float=1)`	✓
Generator	`yield i`	✓
Decorator	`@cache`	✓

1.1 Supported magic methods

Unary operators

- `__repr__`
- `__str__`
- `__hash__`
- `__len__`
- `__iter__`
- `__next__`
- `__neg__`

Logical operators

- `__eq__`
- `__lt__`
- `__le__`
- `__gt__`
- `__ge__`
- `__contains__`

Binary operators

- `__add__`
- `__radd__`
- `__sub__`
- `__rsub__`
- `__mul__`
- `__rmul__`
- `__truediv__`
- `__floordiv__`
- `__mod__`
- `__pow__`
- `__matmul__`
- `__lshift__`
- `__rshift__`
- `__and__`
- `__or__`
- `__xor__`
- `__invert__`

Indexer

- `__getitem__`
- `__setitem__`
- `__delitem__`

Specials

- `__new__`
- `__init__`
- `__call__`
- `__divmod__`
- `__enter__`
- `__exit__`
- `__name__`
- `__all__`

1.2 Builtin functions

```
repr
exit
len
hex
iter
next
hash
abs
divmod
round
isinstance
issubclass
callable
getattr
setattr
hasattr
delattr
chr
ord
id
globals
locals
exec
eval
compile
quit
```

2 Supported modules and functions of the mikroCNC program

To access functions and defined constants, it is necessary to include the appropriate module via the **import** directive. Functions are called using the construct **module.function().**

Code example:

```
import mikrocnc as mcnc
import gcode

print('Loading file...')
ok = gcode.LoadFile('c:\\gcode\\myfile.tap')

if ok: mcnc.DoButton(mcnc.IDC_START)
else: print('Error opening file!')
```

2.1 module **mikrocnc**

Functions:

```
IsBusy() -> bool
WaitIdle()
StatusMsg(text: str)
ErrorMsg(text: str)
Beep()
Sleep(ms: int)
GetSafeZ() -> float
IsSafeZ() -> bool
LinearMove(feed: float, x,y,z,a,b,c:float)  #for rapid, feed = -1
LinearMoveAbs(feed: float, x,y,z,a,b,c:float) #for rapid, feed = -1
LinearMove6(feed: float, p:Point6)
LinearMoveAbs6(feed: float, p:Point6)
IsMoveBusy() ->bool
GetAbsPosition(axis: int) -> float  #axis = 0-5
GetAbsPosition6() -> Point6
GetPosition(axis: int) -> float  #axis = 0-5
GetPosition6() -> Point6
GetSoftMax(axis: int) -> float  #axis = 0-5
GetSoftMin(axis: int) -> float  #axis = 0-5
GetSoftMax6() -> Point6
GetSoftMin6() -> Point6
GetToolChangeStart(axis: int) ->float #axis = 0-5
GetToolChangeStart6() -> Point6
IsHomed(axis: int) ->bool  #axis = 0-5
HomeAxis(axis: int)  #axis = 0-5
HomeCombination(mask: int)
SetCurrentTool(num: int)
ToolTableSet(tool: int, id: int, value: float)
ToolTableGet(tool: int, id: int) -> float
```

```
SpindleStartCW()
SpindleStartCCW()
GetSpindleState() -> int # STOP=0, CW=1, CCW=2
SpindleStop()
CoolantStart(type: int)
CoolantStop()
GetCoolantState() -> int
DoButton(code: int)
MessageBox(type: int, text: str) ->int
CallScript(file_name: str)
GetDRO(id: int) -> float
SetDRO(id: int, value: float)
SetLED(id: int, state: bool)
GetLED(id: int) -> bool
ZeroAxisAbs( axis_mask: int)
SetHomed(axis: int, value: bool)
ActivateESTOP(text: str)
JogOn(axis:int, dir:int, speed:int)
JogOff(axis:int)
SetJogMode(mode:int)
GetJogMode() -> int
GetMacroPath() -> str
GetKeyState(vk_code: int) -> int
SetRetData(value:str|bool|int)
```

2.2 module **gcode**

```
Functions:

LoadFile(path_name: str) -> bool
HasFile() ->bool
GetFileName() -> str
Run()
GetRunState() -> int    # STOP=0, RUN=1, PAUSED=2
StopExec()
ExecLine(text: str)
SetFeedrate(feedrate: float)
GetVar(address: int) ->float
SetVar(address: int, val: float)
GetProbeResult() ->int
GetParameter(par: int) -> float | int
GetMin(axis: int) ->float
GetMax(axis: int) ->float
```

2.3 module **OutputSignal**

```
Functions:

SetState( id: int, state: bool)
GetState( id: int) -> bool
```

2.4 module **InputSignal**

```
Functions:

GetState(id: int) -> bool
GetAnalog(num: int) -> int
GetEncPos(num: int) -> int
```

2.5 module **mcnctypes**

The module contains declaration of the **Point6** class object
This class has attributes that represent 6 coordinates of a point (x,y,z,a,b,c) as well as some basic operations.

2.6 module **profile**

The module allows the use of the currently active profile for permanent saving and later loading of variables. The saved variables are preserved and available upon restarting the script, as well as when restarting the program.

Functions:

```
SaveStr(name: str, value: str, glob:bool=False)
LoadStr(name: str, glob:bool=False) -> str
```

2.7 module **gui**

The module enables implementation of the user interface in the form of dialog box with common windows controls.

Functions:

DialogBox(w: int, h: int, title) -> DialogBox

Class **DialogBox** contains functions:

```
AddButton( x,y,w,h, name, align, offx,offy, label, callback) -> bool
AddCheckBox( x,y,w,h, name, align, offx,offy, label, callback,
default_val) -> bool
```

```
AddEdit( x,y,w,h, name, align,offx,offy,ronly,default_val)
AddText( x,y,w,h, name, align,offx,offy,label) -> bool
AddRadio(self, x,y,w,h, name,align,offx,offy,label,callback, default_val)
-> bool
AddCombo(self, x,y,w,h, name,align,offx,offy,label,callback, default_val)
-> bool
DoModal() ->int
GetValue(name: str) -> item value type (int|bool|str)
SetValue(name: str, value)
EnableItem(name: str, ena: bool)
SetDefaultButton(name: str)
SetMargins(xmarg: int, ymarg: int)
GetRadioValue( name_first: str)
SetRadioValue(name_first:str, value:int)
EndDialog(code: int)
UpdateData()
Delete()
```

3 Calling macros and passing arguments

Macros are called from the MDI line or from G-code using the command Mxxx, where xxx is the macro number. This executes the Python script Mxxx.py

.py scripts are ordinary text files that can be edited using any text editor (such as **notepad.exe** in Windows or the free **Notepad++** which offers advanced options, syntax highlighting, etc.).

When calling an M macro, 4 parameters are supported: L, P, Q and R.

Parameters that are not specified are received by the program as an empty string. The first argument is the name of the macro, followed by the specified 4 parameters.

Example call: “M123 L1 P2 Q3 R4”

program example:

```
import sys
print(‘args=’, sys.argv)
#================================
# result of execution:

args=['M123', '1', '2', '3', '4']
```

NOTE: For safety reasons, it is recommended to test scripts first without connecting the machine or without tools.

3.1 Notes and current limitations

- Currently, only one user-defined M macro can be used in a line of G-code (but it is possible to additionally call standard M commands like **M3, M4**, etc. in the same line).

  It is possible to use one script from another script, that is, to use its functions and objects through the **import** directive. The called code then executes within the same virtual machine. It is also possible to call another script using the function **mikrocnc.CallScript()**, in which case the called script executes in a separate virtual machine.
- The **gcode.ExecLine()** function executes in a separate G-code context, so that executing commands generally does not change the state/parameters defined in the main G-code.

3.2 Macro log window

The **Macro Log window** (Figure 3.1) is used to monitor the output of Python scripts as well as for information about any errors that occur during the execution of scripts. This window can be opened via the option from the menu **Menu/View/Macro log window**.

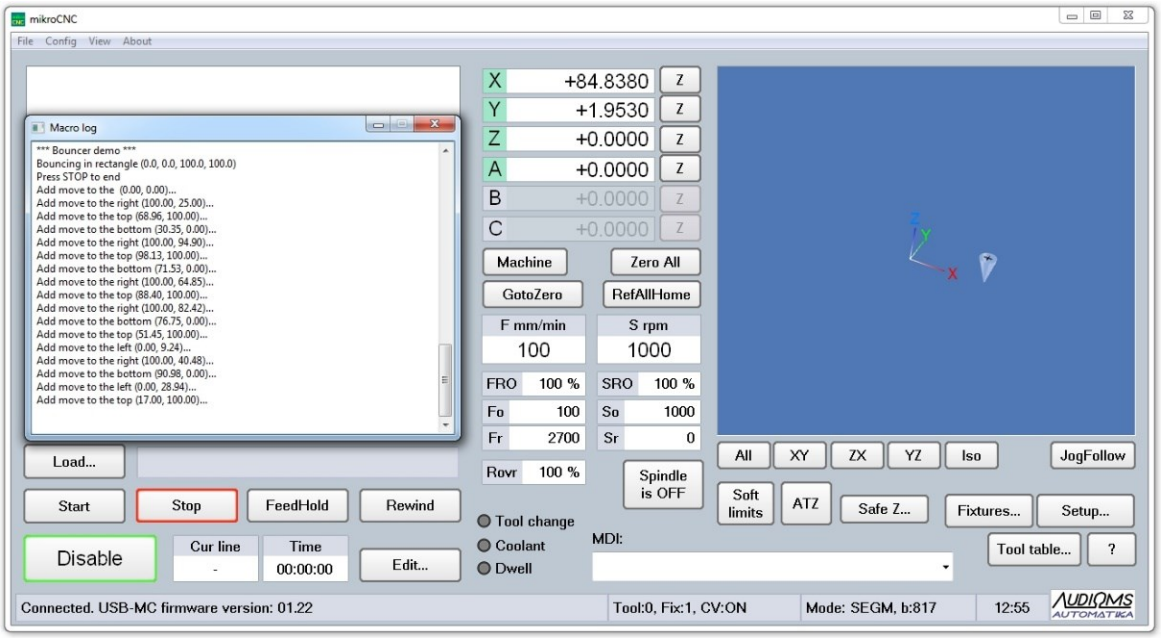


Figure 3.1 Macro Log window

Using the configuration dialog in the **UI options** section, it is possible to set options to automatically open this window in case of an error or when any new message appears.

4 Functions description

4.1 Module mikrocnc

`IsBusy() -> bool`

The function is used to check if any operation is in progress. It can be used for synchronization when it is necessary to know if the machine is ready to initiate new operations.

The return value is of **bool** type: **True** or **False**.

`WaitIdle()`

The function waits for all currently executing operations to complete. Many functions only initiate the execution of certain operations, and control is immediately returned to the caller. It is often necessary to wait for initiated operations to fully complete before issuing new commands.

An example would be the function for executing a line of G-code **gcode.ExecLine()**.

This function internally calls IsBusy() for checking if any operation is in progress.

`StatusMsg(text: str)`

Shows a text message in the status line of the mikroCNC program.

`ErrorMsg(text: str)`

Shows a message in red color in the status line of the mikroCNC program along with a sound signal and blinking.

`Beep()`

Generates a sound signal.

`Sleep(ms: int)`

The function is used to wait for a specified time interval. The time is specified in milliseconds. During the wait period, the script thread does not consume any CPU time, so this function can also be used, for instance, in a loop that would be executed for a longer period of time without needing to be excessively fast, thus saving CPU time for other threads and processes.

`GetSafeZ() -> float`

The function returns the value of the **SafeZ** height. The value returned is in machine coordinates and is obtained by calculation based on the options set by the user in the mikroCNC program in the **SafeZ** settings window.

`IsSafeZ() -> bool`

This function is used to check whether the **SafeZ** option is enabled in the program.

```
LinearMove(feed, x,y,z,a,b,c) #default feed = -1 (rapid), x,y,..= None
LinearMoveAbs(feed, x,y,z,a,b,c) #version for absoulte coordinates
```

From the mikroCNC version 0.71 and later, all parameters are optional and have appropriate default values. Specifying parameters by name is supported e.g. **LinearMove**(x=50, y=100)

feed:float – **feedrate** for the move, **default is -1 (rapid)** if not specified,
x,y,z,a,b,c:float – coordinates of the target point. **Dafault value for every coordinate is None**, in which case there will be no change of position for that axis.

These functions are used to perform linear movement from the current position to the target position specified by the coordinates (x, y, z, a, b, c). In the case of the **LinearMove()** function, the coordinates given are working coordinates in accordance with the currently applied work offsets. In the case of the **LinearMoveAbs()** function, the target point is specified in machine (absolute) coordinates.
The **feedrate** parameter determines the movement speed in mm/min. If the feedrate is set to **-1**, the movement occurs at **G0** (rapid) speed.
These functions initiate the mechanism for generating motion, and then control is returned to the main program of the script. Movement is generated with acceleration and deceleration in accordance with the program settings, and the required data is sent to the motion buffer to be executed by the controller.
Before issuing a new **LinearMove()** command, it is necessary to check using the **IsMoveBusy()** function whether the program is ready to accept a new movement command.
Thus, it is not necessary to wait for all operations to complete as with the **gcode.ExecLine()** command.
Therefore, the resulting movement, when consisting of several linear segments, occurs without pauses between those segments.

```
LinearMove6(feed: float=-1, p:Point6)
LinearMoveAbs6(feed: float=-1, p:Point6)
```

Similar to above described functions, but these functions take a **Point6** object as the argument for target point coordinates. To use **Point6** object it should be included from **mcnctypes** module like shown bellow in the example for **GetAbsPosition6**() function.
Normally coordinates p.x, p.y, p.z, etc. should be numbers, but also **None** type is supported as value in which case there will be no change of position for that axis.

```
IsMoveBusy() ->bool
```

This function is used to check whether it is possible to issue a new motion segment via the **LinearMove()** or **LinearMoveAbs()** command. When the function returns **False**, it means that it is possible to add new data into the motion buffer.

GetAbsPosition(axis: int) -> float #axis = 0-5

The function returns the current absolute position of the given axis (in machine coordinates).

GetAbsPosition6() -> Point6

The function returns the current absolute position of all 6 axes in the form of a class object **Point6** defined in the module **mcnctypes**.

Point6 has attributes: x, y, z, a, b, c

To use the **Point6** type object, it is necessary to include its declaration via the **import** directive.

Example code:

```
import mikrocnc as mcnc
from mcnctypes import Point6
p = mcnc.GetAbsPosition6()
print(p.x, p.y, p.z)
```

GetPosition(axis: int) -> float #axis = 0-5

The function returns the current position for the given axis in working coordinates.

GetPosition6() -> Point6

The function returns the current position of all axes in working coordinates in the form of a **Point6** object.

GetSoftMax(axis: int) -> float #axis = 0-5

The function returns the configured maximum value of SoftLimit for the given axis

GetSoftMin(axis: int) -> float #axis = 0-5

The function returns the configured minimum value of SoftLimit for the given axis

GetSoftMax6() -> Point6

The function returns the configured maximum value of SoftLimit for all axes in the form of a **Point6** object

GetSoftMin6() -> Point6

The function returns the configured minimum value of SoftLimit for all axes in the form of a **Point6** object

GetToolChangeStart(axis: int) -> float #axis = 0-5

The function returns the position of the axis in working coordinates for resuming work after tool change operation. It is used, for example, in the M6 user scripts for automatic tool change support.

GetToolChangeStart6() -> Point6

The function returns the positions of all axes in working coordinates for resuming work after tool change.

IsHomed(axis: int) ->bool #axis = 0-5

The function is used to check if the axis is referenced.

HomeAxis(axis: int) #axis = 0-5

The function is used to start the referencing of one axis. The referencing sequence for the axis is then performed autonomously by the controller. It is possible to immediately start the referencing of additional axis without waiting for the first operation to complete (without calling **WaitIdle**()) if that is appropriate for the machine.

HomeCombination(mask: int)

The function is used to simultaneously start the referencing of multiple axes. The first six bits (bits 0-5) of the mask parameter determine which axes are referenced. Bit 0 represents the X axis, bit 1 the Y axis, and so on.

SetHomed(axis: int, value: bool) #axis = 0-5

The function is used to set or reset homed indication for the axis.
It has the same effect as using **SetLED**() with appropriate identifier for the axis.

SetCurrentTool(num: int)

The function sets the current tool. The index number in the tool table is given. The currently active tool and the newly selected tool number can be obtained via the function **gcode.GetParameter()**

ToolTableSet(tool: int, id: int, value: float)

The function is used to write a value to the tool table.
Parameters:

tool – the tool number (row in the tool table),
id – column to be written to (0=Xoffset, 1=Yoffset, 2=Zoffset, 3=Diameter),
value – value in mm that should be written

ToolTableGet(tool: int, id: int) -> float

The function is used to read a value from the tool table.
Parameters:

tool – the tool number (row in the tool table),
id – column to be read (0=Xoffset, 1=Yoffset, 2=Zoffset, 3=Diameter)

SpindleStartCW()

The function starts the Spindle in the forward direction (clockwise). Equivalent to command M3.
The specified delay on starting is used as with command M3.

SpindleStartCCW()

The function starts the Spindle in the reverse direction (counterclockwise). Equivalent to command M4.
The specified delay on starting is used as with command M4.

GetSpindleState() -> int # STOP=0, CW=1, CCW=2

Returns the current state of the Spindle

SpindleStop()

Stops the spindle. Equivalent to command M5. The specified stop delay for spindle is used.

CoolantStart(type: int) # FLOOD=1, MIST=2, BOTH=3

Starts the cooling systems. The parameter **type** determines which functions are activated.

CoolantStop()

Stops both cooling systems (FLOOD and MIST). Equivalent to command M9.

GetCoolantState() -> int # NONE=0, FLOOD=1, MIST=2, BOTH=3

The function returns the current state of the cooling systems.

DoButton(code: int)

The function is used to call the function of the given program button (as if the user clicked the button).

Button codes

IDC_START	1003
IDC_STOP	1004
IDC_FEEDHOLD	1005
IDC_SPINDLE	1007
IDC_RESET	1015
IDC_ZEROX	1058
IDC_ZEROY	1061
IDC_ZEROZ	1062
IDC_ZEROA	1063
IDC_ZEROB	1064
IDC_ZEROC	1065
IDC_ZEROALL	1067
IDC_REFALL	1022
IDC_ATZ	1110
IDC_SOFTLIMITS	1066
IDC_THCON	1035
IDC_THCMINSPEEDON	1042

GetDRO(id: int) -> float

The function returns the value read from the specified DRO field.

SetDRO(id: int, value: float)

The function is used to write a desired value to the specified DRO field.

Defined constants for IDs of DRO fields

IDC_FEEDRATE	1001
IDC_SPINDLESPEED	1002
IDC_FRO	1033
IDC_RRO	1034
IDC_SRO	1032
IDC_X	1011
IDC_Y	1012
IDC_Z	1013
IDC_A	1020
IDC_B	1026
IDC_C	1028
IDC_THCSPEED	1036
IDC_THCVNOM	1037
IDC_THCV	1039
IDC_THCMAX	1040
IDC_THCMIN	1038
IDC_THCCUR	1041
IDC_THCMINSPEED	1043
IDC_CURLINE	1018

NOTE: **Reading X, Y, Z, etc. DRO values will always return work coordinates even if the machine coordinates mode is currently turned ON.**
Also, setting a value to one of these DRO fields will set work coordinate for the current position, in other words the work offset for the axis will be changed.

SetLED(id: int, state: bool)

Sets state for the LED indicator or button illumination indicator.

This functions also activates/deactivates related function for the button, for example:

```
SetLED(mikrocnc.IDC_SOFTLIMITS, True)
```

would turn the Soft Limits on.

GetLED(id: int) -> bool

Returns the current state of the LED indicator or button illumination indicator.

Defined constants for LED indicators

IDC_X	1011
IDC_Y	1012
IDC_Z	1013
IDC_A	1020
IDC_B	1026
IDC_C	1028
IDC_SOFTLIMITS	1066
IDC_THCON	1035
IDC_THCMINSPEEDON	1042

NOTE: IDC_X, IDC_Y, IDC_Z, etc. can be used to set the axis referenced indicators by **SetLED**() function.

```
ZeroAxesAbs( axis_mask: int)
```

This function is used to zero (reset) machine coordinates for the selected axes. Its intended use is for homing scripts to enable advanced homing procedures. Changing machine coordinates in normal work flow regime is **not recommended**.

NOTE: The axis coordinates are reset to the home offsets that are specified in the homing setup.

axis_mask - bit mask that defines all axes for which machine coordinates should be reset. Bit 0 should be set for X, bit 1 for Y, etc.

```
ActivateESTOP(text: str)
```

Activates the safety **ESTOP** mode with a message that is displayed in the program's status bar. It is most commonly used in the event of an error. Activating ESTOP mode results in stopping all operations including all scripts currently executing. This means that after this function is called, the calling script also finishes execution.

```
CallScript(file_name: str)
```

Calls the execution of another script with the given name. The called script runs in an independent virtual machine. The function returns only when the called script has finished executing.

```
JogOn(axis:int, dir:int, speed:int)
```

axis: axis to jog, 0-5

dir: direction, 0=forward, 1=backward

speed: motion speed given as a percentage of the maximum speed of the axis

In case that continuous Jog mode is active, this function initiates the Jog motion of one axis. The motion features acceleration and deceleration according to the currently set profile in the mikroCNC configuration. It is possible to initiate simultaneous motion of

multiple different axes by using multiple calls to the **JogOn**() function for each axis, and the axes initiated this way move independently one of each other.
Additionally, it is possible to change the speed of the Jog motion of an axis by issuing a new **JogOn**() command for that axis with a new speed. The transition from the old speed to the new one features acceleration/deceleration according to the specified profile.
In case the step jog mode (step by step) is active, calling this function executes one step. The step value is set through the mikroCNC **Jog&Mpg dialog** (Jog step [mm]).

`JogOff(axis:int)`

This function stops the Jog for the given axis (axis=0-5). The stopping is carried out with deceleration according to the currently set profile. This function is only used in continuous Jog mode.

`SetJogMode(mode:int)`

The function is used to set operation mode for Jog.
mode=0 : continuous Jog, **JogOn**() starts continuous movement,
mode=1 : step by step, each call to **JogOn**() executes one defined step,
mode=2 : MPG mode

`GetJogMode() -> int`

This function returns the currently active Jog mode. For the meaning of the return value, see the previous function.

`GetMacroPath() -> str`

The function returns the complete disk path for the macro folder of the currently active profile.

`MessageBox(type: int, text: str) ->int`

This function enables displaying of a small window with a message (the Windows system MessageBox function is used internally) and the return value of the function indicates which button the user pressed. It is possible to choose various MessageBox types, icon types, and query types, i.e., buttons that the user can select.
Parameters:

type is formed using a bitwise OR operation (|) of defined constants for three parameters: MessageBox type, icon type, and default button:
type = MessageBox_type | MessageBox_Icon_type | Default_button

text – is the text that is displayed as a message to the user
The return value is the **ID** of the button that the user pressed to exit.

MessageBox types (hex values)

MB_OK	0x00000000
MB_OKCANCEL	0x00000001
MB_ABORTRETRYIGNORE	0x00000002
MB_YESNOCANCEL	0x00000003
MB_YESNO	0x00000004
MB_RETRYCANCEL	0x00000005

MessageBox Icon type (hex values)

MB_ICONHAND	0x00000010
MB_ICONQUESTION	0x00000020
MB_ICONEXCLAMATION	0x00000030
MB_ICONASTERISK	0x00000040
MB_ICONWARNING	MB_ICONEXCLAMATION
MB_ICONERROR	MB_ICONHAND
MB_ICONINFORMATION	MB_ICONASTERISK
MB_ICONSTOP	MB_ICONHAND

MessageBox Default button (hex values)

MB_DEFBUTTON1	0x00000000
MB_DEFBUTTON2	0x00000100
MB_DEFBUTTON3	0x00000200
MB_DEFBUTTON4	0x00000300

Return values

IDOK	1
IDCANCEL	2
IDABORT	3
IDRETRY	4
IDIGNORE	5
IDYES	6
IDNO	7

GetKeyState(vk_code: int) -> int

This function returns the state of a keyboard key. The implementation internally uses Windows system function with the same name. Obtaining a key status is possible when mikroCNC is currently active application (is not minimized). Otherwise this function returns 0.

vk_code – system key code for the key to check

Some of the key codes:
VK_F1 = 0x70
VK_F2 = 0x71
VK_F3 = 0x72
VK_F4 = 0x73
VK_F5 = 0x74
VK_F6 = 0x75
VK_F7 = 0x76
VK_F8 = 0x77
VK_F9 = 0x78
VK_F10 = 0x79
VK_F11 = 0x7A
VK_F12 = 0x7B
VK_CTRL = 0x11
VK_SHIFT = 0x10

Function return value:

- If bit 15 is set (1), key is pressed; otherwise is not pressed.
- if bit 0 is set (1), key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the bit 0 is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

```
#example function to check if a key is pressed
def IsKeyDown(vkey):
    return 0!=(mcnc.GetKeyState(vkey) & 0x8000)
```

SetRetData(value:str|bool|int|float)

This function is used to set the script result return value. Value can be of type str, bool int or float.
With a certain functions (like axis calibration wizard and OnOK scripts) the mikroCNC main program relies on this data for further operation after executing the script.

4.2 Module gcode

Functions:

`LoadFile(path_name: str) -> bool`
This function is used to load a G-code file with the given name. If the loading is successful, the return value is **True**; otherwise, it is **False**.

`HasFile() ->bool`
This function returns True if a G-code file is loaded; otherwise, it returns **False**.

`GetRunState() -> int # STOP=0, RUN=1, PAUSED=2`
This function returns the current run state of the loaded G-code program.

`GetFileName() -> str`
Returns the name and full path of the currently loaded G-code file if it exists. Otherwise, it returns an empty string.

`Run()`
This function starts the execution of the loaded G-code program as if the START button is pressed.

`StopExec()`
This function stops the execution of the G-code program. Further execution of the G-code will be halted when the line currently being executed completely finishes its operations.

`ExecLine(text: str)`
The function executes a line of G-code specified in the form of a string **text**. The string **text** can contain all supported G and M commands.

This function only initiates the operation execution mechanism, and control is returned to the caller, namely the main program of the script, once all commands have been sent for execution. This does not necessarily implies that all given operations are completely finished.
Before assigning a new command **ExecLine**(), to ensure that all previously lounched operations are completed, it is necessary to use the **mikrocnc.WaitIdle()** function.
If the given line of G-code contains a call to some other user script, upon its execution further processing of other commands in the line will be delayed until the script finishes its work.

`SetFeedrate(feedrate: float)`
This function is used to set the feedrate parameter.

```
GetVar(address: int) ->float
```
This function is used to read the value of a given G-code variable. G-code variables in the G-code program are denoted by **#variable_address**.

```
SetVar(address: int, val: float)
```
This function is used to assign a value to the specified G-code variable. The value is of type **float**

```
GetProbeResult() ->int  #see defined constants for return value
```
This function is used to obtain the result/state of a previously initiated **Probe** operation. In case the Probing is successfully completed (return value is PROBE_HIT), the G-code variables **#2000-#2005** contain the coordinates of the contact point. The variables are:

- 2000 -> value of the X coordinate,
- 2001 -> value of the Y coordinate,
- 2002 -> value of the Z coordinate,
- 2003 -> value of the A coordinate and
- 2004 and 2005 not supported currently.

Return values for GetProbeResult()

PROBE_IDLE = 0	No Probing operation was issued
PROBE_ACTIVE = 1	Probing is in progress
PROBE_NOHIT = 2	Probing ended unsuccessfully without contact
PROBE_HIT = 3	Probing is finished successfully

```
GetParameter(par: int) -> float | int  #see defined constants
```

Defined constants for GetParameter()

SPINDLE_SPEED	0
FEEDRATE	1
CIRC_MODE	2
WORK_PLANE	3
CV_MODE	4
IJK_ABS	5
TOOL_OFFSET_NDX	6
TOOL_TABLE_NDX	7
METRIC_UNITS	8
RELATIVE_MODE	9
CURRENT_TOOL	10
SELECTED_TOOL	11

```
GetMin(ax: int) ->float
GetMax(ax: int) ->float
```

Functions are used to get the overall dimensions of the loaded G-code. The **GetMin(ax)** function returns the smallest grid loading coordinate along the axis specified via the **ax** parameter. Similarly, the **GetMax(ax)** function returns the maximum coordinate of the loaded mesh for the given axis.

ax – axis number: 0 = x, 1 = y, 2 = z

4.3 Module OutputSignal

Functions:

```
SetState( id: int, bool state)
```

The function is used to set the output signal state. The signal identifier is specified (see table) and the desired new state, active or not (**True**/**False**).

Note: In order for the signal state to be linked to an output pin of the motion controller, it is necessary to enable the given signal in the configuration (Config/Setup/Output signals) and set the desired output pin, as well as other options if needed.

```
GetState( id: int) -> bool
```

The function is used to obtain the current state of the output signal. The parameter **id** is the signal identifier (see table).

Defined constants for id of output signals

EXT1	0
EXT2	1
EXT3	2
EXT4	3
EXT5	4
EXT6	5
ENABLE	6
OUTPUT1	9
OUTPUT2	10
...	
OUTPUT10	19

4.4 Module InputSignal

Functions:

`GetState(id: int) -> bool`

The function returns the current state of the given input signal. The parameter **id** is the signal identifier (see table). The return value is **True** or **False** (active or inactive).

Note: For the signal state to reflect the state of a certain input pin of the motion controller, it is necessary to enable the signal in the configuration (Config/Setup/Input signals) and set the desired input pin, as well as other options if needed.

`GetAnalog(num: int) -> int`

The function returns the current voltage value on the specified analog input obtained via the A/D converter. The value obtained is a 16-bit unsigned integer, meaning that 65535 corresponds to the maximum possible voltage value (5V).

Defined constants for id of input signals:

HOMEX	0
LIMITXP	1
LIMITXM	2
HOMEY	3
LIMITYP	4
LIMITYM	5
HOMEZ	6
LIMITZP	7
LIMITZM	8
HOMEA	9
LIMITAP	10
LIMITAM	11
HOMEB	12
LIMITBP	13
LIMITBM	14

HOMEC	15
LIMITCP	16
LIMITCM	17
ESTOP	18
INDEX	19
PROBE	20
THC_ARCOK	21
THC_UP	22
THC_DOWN	23
INPUT1	24
INPUT2	25
...	...
INPUT10	34

`GetEncPos(num: int) -> int`

The function returns the current counter value for the given encoder. The value is a 32-bit integer. The parameter **num** is the position of the encoder/MPG in the settings table in the **mikroCNC** program, starting from 0.

Note: The refresh period for the state of all inputs, including analog inputs and encoders, is 50ms.

4.5 Modul mcnctypes

The module contains the definition of the **Point6** class, which has attributes **x, y, z, a, b, c** that can be considered to be coordinates of a point (for all 6 axes) or 6 components of a vector.
The class also contains the following functions:

`__init__(self, x=0,y=0,z=0,a=0,b=0,c=0)`
The class constructor for initializing coordinates.

`__add__(self, p: Point6) ->Point6`
The addition operator allows vector addition of two **Point6** class objects.

`__sub__(self, p: Point6) ->Point6`
The subtraction operator allows vector subtraction of **Point6** class objects.

`__str__(self) -> str`
String representation operator provides all coordinates in the form of a printable string.

`__mul__(self, a) -> Point6`
The muliplication operator allows mulitiplication of vector **Point6** by a scalar **a**.

`__div__(self, a) -> Point6`
The division operator allows division of **Point6** vector components by a scalar **a**.

`Res(self) -> float`
Function returns the resultant of vector **Point6**.

`Distance(self, p:Point6) -> float`
Function returns the distance between two **Point6** points.

4.6 Modul profile

`SaveStr(name: str, value: str, glob:bool=False)`

The function saves value of a string variable to the current .INI profile. A record is created in the profile file in the form **name = value.**
If **glob** parameter is ommited or set to **False**, the record is saved in a special group that is created for the active script. This data section is private for the script so that every script can manipulate only with its own recorded data.
If **glob** parameter is set to **True** then the record is saved in the global section that is shared between all scripts and all scripts in this profile can read and modify that data.

name – identifier to use for saving the value. Name is a string that can contain letters, numbers and underscore character '_'
value – value that is to be recorded (string)
glob – optional global flag that indicates wether the value is saved in the data section private for this script (glob=False) or in the global section so that other scripts can read/write this data (glob=True).

```
LoadStr(name: str, glob:bool=False) -> str
```

Function reads and returns value of a variable that is saved in the current profile.

name – identifier that is used for saving the value
glob – optional flag that indicates wether the value is loaded from the data section private for this script (glob=False) or from the global profile section (glob=True)

4.7 Module gui

Module **gui** enables implementation of the user interface in a form of dialog box that can contain some basic Windows controls for the interaction with user. (Figure 4.1).

Currently supported elements that can be show in a dialog box are:

- edit field
- button
- radio button
- check box
- combo box
- text

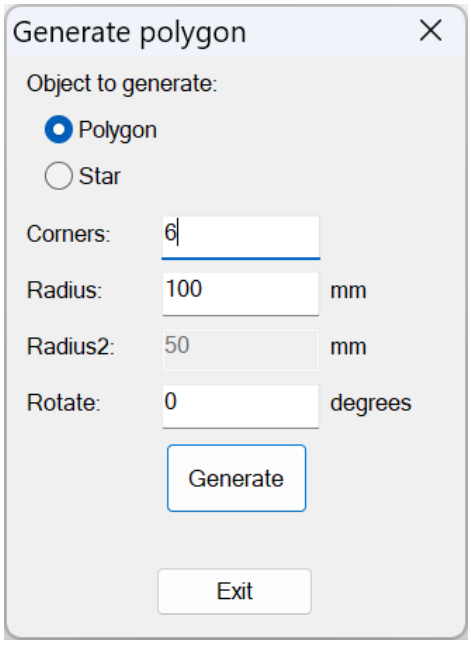


Figure 4.1 Example dialog box created from the script (example M116)

The implementation of the **gui** module is designed to enable as simple as possible application with the least amount of code. For this purpose, most functions have default parameter values defined, allowing only the necessary parameters to be specified (arguments can be specified by name rather than by order).

Additionally, it is usually unnecessary to specify absolute coordinates and dimensions of the elements as automatic positioning of the element on the dialog is ensured using available alignment, offset, and margin adjustment methods.

The object (class) **DialogBox** is created as follows:

```
d = gui.DialogBox(w:int,h:int,title:str)
#variable d now represents a DialogBox object
```

Parameters (all optional):

- **w** – width of the dialog box work area (does not include frame and title),
- **h** – height of the dialog box work area (does not include frame and title),
- **title** – title for the dialog box.

Here, width and height (as well as coordinates and all dimensions in the further text) are specified in DLU (Dialog Logical Units) as is usual for Windows dialog resources (they are not pixels, but these units are derived from the dialog font size).
DLU units are adopted so that the created dialog looks correct and approximately the same on every system.
It is also important to note that in this function, **w** and **h** refer to the width and height of the **dialog's client area**. This facilitates planning the positions of elements since the width of the frame and title, which may vary from one system to another, does not have to be taken into account.

The **DialogBox()** function returns an object through which further creation of the dialog's content is performed. The dialog box is still not visible on the screen at this point and will become visible only after all initialization is finished and the function **DoModal()** is called.

class DialogBox object has the following member functions:

Functions for creating elements (Add*) expect all coordinates and dimmensions in the Windows DLU units (Dialog Logical Unit).
All element types have **name** (string) identificator for later referencing (if needed), **align** (Int) value that specifies alignment type, **offx** and **offy** (Int) offsets for optional additional position correction relative to the previous element. Most elements feature a **label** string that specifies text label for the elemennt.

AddButton(x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, label:str, callback:str) -> bool

The function creates a button. **All parameters are optional** and the parameters are:

- **x,y** – coordinates of the upper left corner

- **w,h** – width and height of the button
- **name** – identifier, necessary if later referencing is needed
  There are special predefined values for identifiers for the OK and Cancel buttons, which are ‘IDOK’ and ‘IDCANCEL’ respectively. If these identifiers are used, there is an automatic mechanism to close the dialog with a return code, so it's not necessary to provide a callback function to handle this.
- **align** – alignment type:
  - **gui.LEFT** – horizontal alignment to the left edge of the window (default)
  - **gui.CENTER** – horizontal centering across the width of the window
  - **gui.RIGHT** – alignment to the right edge of the window
  - **gui.LEFTPREV** – alignment relative to the previously created element

  The first three types of alignment imply that the element is created in a new row (below the previous element). Only the last value (gui.LEFTPREV) locates the element in continuation of the same row, i.e., next to the previous element on its right side.
  Note: The alignment type **gui.LEFTPREV** aligns the given element next to the previous one so that these elements (two or more consecutive) form a group. This whole group of elements is aligned in the dialog as specified for the first element of the group (for which the alignment is not of type **gui**.**LEFTPREV**).
  In this way, it is possible, for example, to horizontally center multiple buttons (a common example is **OK** and **Cancel** buttons centered at the bottom of the dialog box).
- **offx, offy** – offset for the relative positioning in relation to the previuos element. If a specific absolute coordinate x or y is given, the offset for that axis is ignored. Default values are 0.
- **label** – text that will be displayed on the button
- **callback** – name of the function that will be called when the button is clicked (if needed).

Example of creating a dialog box and one button with a callback function (Figure 4.2):

```
import gui

def on_click():
    print(‘Button clicked!’)

d = gui.DialogBox(150,100,’Hello world!’) #create new DialogBox
d.AddButton(name=’button1’, label=’Test’, w=50, callback=’on_click’)
#Since coordinates are not specified, the button will be created in the upper
left corner.
d.DoModal() #show dialog box
```

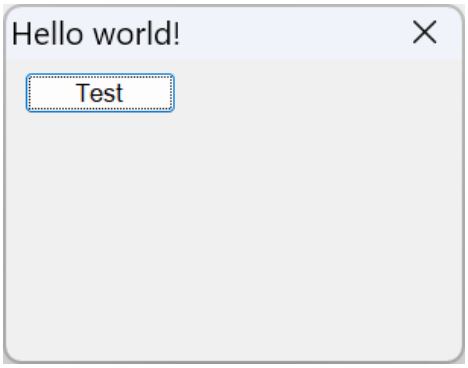


Figure 4.2

Pressing the button will trigger the **on_click()** function and a message will be printed in the macro log window.

When adding each subsequent dialog element, if the x and y coordinates and alignment specifier are omitted, the new element would be created in the next row (here below the button) and aligned to the left edge of the window.

So the default positioning is: next row and left alignment.

The following is an example of creating a group of centered elements (in this case three buttons) in a way that is explained in the note for the **align** parameter (Figure 4.3).

```
import gui

d = gui.DialogBox(200,100,'Hello world!') #create a new DialogBox
#the first element of the group determines the group alignment in the window
d.AddButton(y=80, name='IDOK', label='OK', w=50, align=gui.CENTER)
#the remaining elements of the group follow (alignmet type LEFTPREV)
d.AddButton(name='apply', label='Apply', w=50, align=gui.LEFTPREV)
d.AddButton(name='IDCANCEL', label='Cancel', w=50, align=gui.LEFTPREV)
d.DoModal() #show the dialog box
```

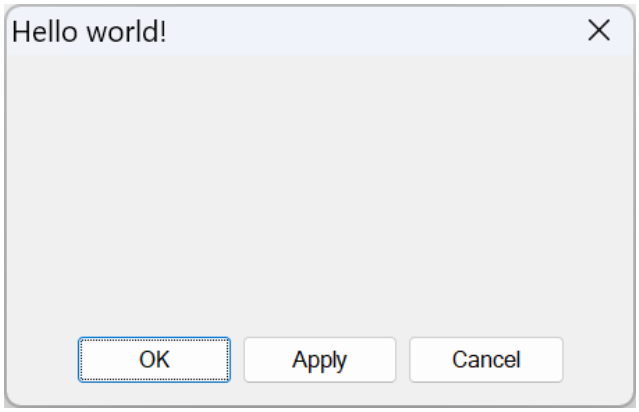


Figure 4.3 Example of creating a centered group of elements

AddCheckBox(x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, label:str, callback:str, default_val:bool) -> bool

The function creates a check box. All parameters are optional.
In addition to the parameters already described (see the **AddButton** function), it is also possible to specify a default value:

- **label** – text that will be displayed next to the checkbox
- **default_val** – determines whether the option will be enabled (True) or disabled (False) when the dialog appears (default is False).

AddEdit(x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, ronly:bool, default_val:str) -> bool

The function creates an edit field. All parameters are optional. In addition to the described parameters, the following are also available:

- **ronly** – ife **True** a read-only edit field is created (default is False)
- **default_val** – the initial value (string) displayed in the field

The Edit element only supports string values. Other variable types (int, float) must be converted to a string, for example, using the Python function **str(value)**.

AddText(x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, label:str) -> bool

The function creates a text.

- **label** - text

Example of creating an edit field with text for description and for unit (Figure 4.4):

```
import gui

#create new DialogBox
d = gui.DialogBox(150,100,'Hello world!')
#create text for description
d.AddText(label='Value:')
#create edit field following the text
d.AddEdit(name='edit1', default_val='123', align = gui.LEFTPREV)
#create text for unit that follos
d.AddText(label='mm', align=gui.LEFTPREV)

ret=d.DoModal() #display dialog

if(ret==1): #OK/Enter pressed
 print('The value of the field is', d.GetValue('edit1'))
```

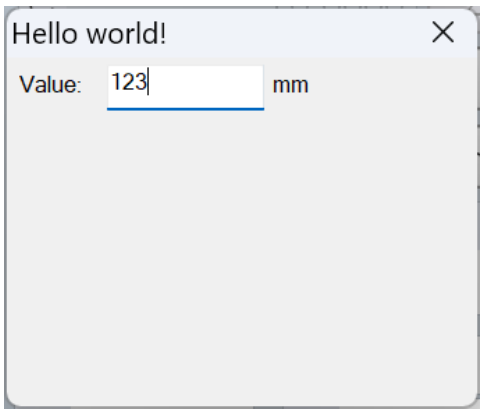


Figure 4.4

```
AddRadio( x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, label:str, callback:str, default_val:bool) -> bool
```

The function creates a radio button.

- **default_val** – initial state, selected or not (default is False)

Multiple consecutively created radio elements form a group, so when the user selects one button, the others are deselected.
Checking the state of individual elements is possible through the **GetValue()** function. It is also possible to obtain the index of the selected element of the radio group using the **GetRadioValue()** function.

```
AddCombo( x:int, y:int, w:int, h:int, name:str, align:int, offx:int, offy:int, label:str, callback:str, default_val:int) -> bool
```

The function creates a combo box that is a type of drop-down list.

- **label** – the parameter is a string that contains the elements of the drop-down list separated by '\n' as a delimiter. For example: the string '**one**\n**two**\n**three**' specifies three elements: '**one**', '**two**', and '**three**'
- **callback** – optional name of the function that will be called when the selected list element changes
- **default_va**l – optional initially selected list element starting from 0

```
DoModal() ->int
```

After creating all the dialog elements and possibly calling other formatting functions, it is necessary to call the **DoModal()** function so that the prepared dialog window is displayed on the screen.
This function also performs the so-called modal loop, meaning it services all dialog controls and does not return until the user closes the dialog box. This can be done, for example, by clicking the **OK** or **Cancel** button or in another way that triggers the **EndDialog()** function.

Upon closing the dialog, the function returns a code that indicates how the dialog box was closed.

-1- error creating the dialog
0 - function failed
1 - OK button
2 - Cancel button
3 - an error occurred

The return code can also be another value passed through the **EndDialog**() function.

```
GetValue(name:str) -> bool/int/str
```

The function returns a value associated with the given dialog box element.

name – identifier of the element.

The type of return value depends of the element type:

- **Edit field** -> str, string entered in the field. This string can be converted as needed to the appropriate type (**int**, **float**) using Python functions **int(str)**, **float(str)**
- **Check box** -> bool, True or False depending on whether the box is checked or not
- **Radio dugme** -> bool, True or False depending on whether it is selected or not
- **Combo box** -> int, index of the selected option starting from 0

```
SetValue(name:str,value:bool/int/str)
```

The function sets a value for the given element.
name – identifier of the element
value – type of value depends on the element type (see previous function)

```
EnableItem(name:str,ena:bool)
```

The function sets the enable state for the elemnent (enabled/disabled)
name – the element identifier
ena – True or False

```
SetDefaultButton(name:str)
```

The function sets the default button for the dialog box. When the ENTER key is pressed, this button is activated.

name – button identifier

SetMargins(xmarg:int, ymarg:int)

The function sets the margins of the dialog box work area for the x-axis (left and right) as well as for the y-axis (top and bottom). Both parameters are optional and are specified in DLU units.
These margins are used for automatic positioning of elements in the dialog's working area.
When an absolute coordinate of the element (x or y) is given, the margins for that axis are not applied.

GetRadioValue(name_first:str)

The function returns the index of the selected option in the group of radio elements.

name_first – identifier of the first element in the group of radio elements

SetRadioValue(name_first:str, value:int)

The function is used to select one option in the group of radio elements. Other options in the group are cleared.

name_first – identifier of the first element in the group of radio elements
value – is zero based index of the option to select

EndDialog(code:int)

The function causes closure of the dialog box with the given return code. The script receives this code as the return value of the **DoModal**() function.
It is typical for the **EndDialog**() function to be called, for example, from a callback function when a button is clicked.
As already noted, the **OK** and **Cancel** buttons do not require a callback function for invocation of **EndDialog**() function in case when predefined identifiers '**IDOK**' and '**IDCANCEL**' are used.

UpdateData()

This function is used to fetch the values of all elements of the dialog box window and store them in the internal memory of the objects before the window is closed. This must be done before calling the **EndDialog**() function. In case of the **OK** button, this function is called automatically.

`Delete()`

This function is used to explicitly free the memory of the dialog box when the work with the dialog is finished. All resources are released anyway when the script finishes running, so if the script opens one (or a smaller number of) dialogs, then calling the **Delete**() function is optional. After calling the **Delete**() function, that dialog box object can no longer be used.

5 Special Purpose Scripts

In the **mikroCNC** program, for multiple functions it is possible to use user scripts tailored for specific purposes. It is possible to redefine the functions of several buttons so that instead of the built-in functionality of the program, a user script is called and executed.

Additionally, for several other functions it is possible to configure user scripts to be called.

Special scripts currently available for customization:

- The **M6** script is used for automatic tool change and is called from G-code when the third option for Automatic Tool Changer is selected in the settings,
- **AutoToolZero** script can be called by pressing the **ATZ** button for automatic tool hight calibration,
- The **RefAllHome** script can be called by pressing the **RefAllHome** button to reference the machine,
- The **M1030** script can be called at the end of the execution of the G-code program after the execution of the **M30** command, and
- **BackgroundTask** script can be enabled for continuous execution in the background.

6 Script Examples

With the **microCNC** program come example scripts that demonstrate a various functions. These examples are located in the folder **macro_examples**.

M100

Example of using the function **gcode.ExecLine**() for generatiing a spiral movement in a shape of the square whose base is gradually decreasing.

M101

Example of using the function **mikrocnc**.**LinearMove**() for moving back and forth with a change in movement speed (**feedrate**)

M102

Example of using the function **mikrocnc**.**LinearMove**() for movement within a given square frame while bouncing off the walls (edges) of the square.

M103

Example of using the **mikrocnc**.**MessageBox**() function to display information and receive feedback from the user.

M104

Example of using the **fileio** module for writing and reading files with the help of the **File** class functions **fileio**.**open**(), **File**.**writeline**(), **File**.**readline**().

M105

Example of controlling the Spindle using the functions **mikrocnc**.**SpindleStartCW**(), **mikrocnc**.**SpindleStartCCW**(), **mikrocnc**.**SpindleStop**().

M106

Example of using the function **gcode**.**LoadFile**() to load g-code and to start it via **mikrocnc**.**DoButton**().

M107

Test of recursive script calling. In addition to using the **ExecLine**() function to start a new instance of the script, it also demonstrates passing arguments to the script and reading them via the **sys** module.

M108

Example of checking whether the referencing of the machine axes has been completed, displaying status via **mikrocnc**.**MessageBox**(), and generating a **RuntimeError**() if appropriate.

M109

Programmable starting of a Jog movement for the axes using the functions **mikrocnc**.**JogOn**(), stopping via **mikrocnc.JogOff()** and with changing the motion speed during the motion.

M110

Example of manual control of Jog movement via external signals that are read using the function **InputSignal**.**GetState**()

M111

Example of reading the connected encoder using the function **InputSignal**.**GetEncPos**()

M112

Example of starting the machine referencing, first only the Z axis using the function **mikrocnc**.**HomeAxis**(), and then all other axes simultaneously using the function **mikrocnc**.**HomeCombination**()

M113

The script generates a G-code file with a mesh in shape of a 3d wave and then this G-code is automatically loaded via the function **mikrocnc.LoadFile()**

M114

The script demonstrates the use of the **profile** module and the saving and loading of variables into/from the current .INI profile using the functions **profile**.**SaveStr**() and **profile**.**LoadStr**()

M115

Example demonstrating the use of the function **os**.**PlaySound**() to play a sound signal from a .WAV file

M116

Example of using the **gui** module to create a user interface in the form of a dialog **gui**.**DialogBox**(). The script generates a polygon or star with given dimensions and a given number of vertices, and then loads this G-code into mikroCNC.

To create dialog box elements, functions of the **DialogBox** object such as: **AddText()**, **AddRadio()**, **AddEdit()**, etc., are used.

M117

A simpler example of using the gui module to open a dialog for data input. In addition to functions for creating dialog elements, the function GetValue() is demonstrated for reading the entered values of variables.

7 G-Code variables

7.1 Defined variables

G-code variables are referred to using a hash (#) symbol in G-code program.

Currently defined G-code variables that are used and maintained by mikroCNC:

2000 - 2003 probe tip contact coordinates (x,y,z,a) after successful probing operation G31

5211 - 5216 temporary offsets (x,y,z,a,b,c) that are set by G92/G52 command

5220 – current fixture number

5221 - 5226 G54 fixture offset values (x,y,z,a,b,c)
5241 - 5246 G55 fixture offset values
5261 - 5266 G56 fixture offset values
...
5601 - 5606 20th fixture offset values

7.2 Persistent variables

User variables in the range 500-599 are persistent in a way that their values are preserved after the program restart.

DOCUMENT REVISION:

- Ver. 1.0, December 2024, Initial version
- Ver. 1.1, March 2025, Added new supported function
- Ver. 1.5, June 2025, Added description of 'gui' module (Chapter 4.7)
- Ver. 1.51, September 2025, Described of functions added in software mikroCNC version V0.71
- Ver. 1.73, December 2025, Described of functions added in software mikroCNC version V0.73