LAUTERBACH
DEVELOPMENT TOOLS

# Controlling TRACE32 via Python 3

Release 02.2024

MANUAL

# Controlling TRACE32 via Python 3

# Controlling TRACE32 via Python 3

**Version 04-Mar-2024**

## History

| | |
|---|---|
| 20-Aug-20 | Manual was updated to introduce new lauterbach.trace32.rcl solution. The ctypes solution became legacy. |
| 23-Dec-16 | Initial version. |

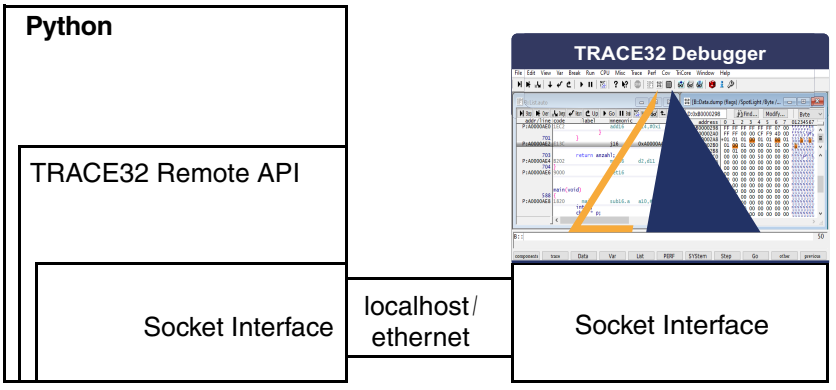
# About this Manual

This document provides information on how Python can be used to control TRACE32.

Please direct questions and feedback to **python-support@lauterbach.com**.

# Introduction

TRACE32 PowerView can be controlled by Python via the TRACE32 Remote API **"API for Remote Control and JTAG Access in C"** (api_remote_c.pdf).




The following options to use the TRACE32 Remote API via Python can be found in your TRACE32 installation:

- `~~/demo/api/python/rcl` contains the Python package lauterbach.trace32.rcl, which will be abbreviated PYRCL in this document. It's available from the DVD.2020.09 and it is recommended for new projects.
- `~~/demo/api/python/legacy` contains Pythons demos on how to use ctypes to load and use the Remote API library (DLL) provided by Lauterbach.

# PYRCL versus TRACE32 Legacy Approach

**lauterbach.trace32.rcl (PYRCL) for Python 3.6+**

We recommend using PYRCL for new projects because:

- It requires less implementation effort.
- It is faster, since it is a native implementation of the RCL protocol.
- It is less error-prone since PYRCL is automatically tested and deployed.

**TRACE32 legacy aproach for Python 3**

As the legacy approach is more a set of examples based on the C implementation of the RCL protocol, it will continue to work and be supported. In some scenarios, it might makes sense to still use the legacy approach:

- To extend or modify existing projects.
- The used Python version is not supported by PYRCL.
- Features, which are not supported by PYRCL, are needed.

# lauterbach.trace32.rcl (PYRCL)

lauterbach.trace32.rcl is compatible with Python 3.6+.

From DVD 2020.09 Lauterbach provides a Python module called **"lauterbach.trace32.rcl"**. This module provides a native Python interface to use the TRACE32 Remote API.

PYRCL supports the TRACE32 Remote API (RCL) in TCP and UDP mode. TCP is recommended. The config.t32 must have one or both of the following blocks:

**TCP (recommended):**

```
…

RCL=NETTCP
PORT=20000

…
```

**UDP:**

```
…

RCL=NETASSIST
PACKLEN=1024
PORT=20000

…
```

## Versioning

PYRCL versions follows **[https://www.python.org/dev/peps/pep-0440]**.

This means:

- PYRCL versions take the form "X.Y.Z". X is the major version, Y is the minor version and Z is the patch version. Pre-releases are denoted with an additional aN (alpha), bN (beta) or rcN (release candidate), with N > 0.
- Major versions introduce backwards incompatible changes to the API. A TRACE32 update will be required and existing scripts might need to get adjusted.
- Minor versions introduce backwards compatible features to the API. A TRACE32 update is recommended.
- Patch versions introduce backwards compatible bug fixes.
- Version 1.0.0 was released with the DVD 2020.09.

## Package

The package is located in your TRACE32 installation under `~~/demo/api/python/rcl`.

The package consists of:

- `~~/demo/api/python/rcl/dist` contains the source and wheel of the package.
- `~~/demo/api/python/rcl/doc` contains the package documentation including examples.

## Documentation

The package is documented in the package itself in the form of docstrings.

Additionally, a HTML documentation is generated which can be found in `~~/demo/api/python/rcl/doc/html/index.html`.

# TRACE32 Legacy Approach

Compatible with Python 3.

Before DVD 2020.09 the only way to use the Remote API was using the Python module ctypes. "ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python."
**[https://docs.python.org/3/library/ctypes.html]**.

## TRACE32 already Started

The Python script below shows a typical command sequence

- That establishes the communication between Python and a debug device.
- That releases the communication between Python and a debug device.

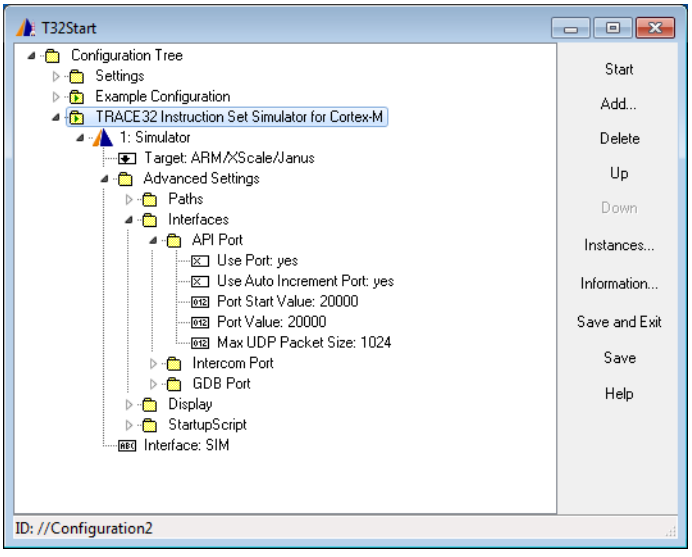
The example assumes the following:

- You are working on a 64-bit Windows system.
- You are using a TRACE32 debugger or a TRACE32 Instruction Set Simulator as debug device.
- The TRACE32 PowerView GUI for the debug device is already running on the same host and is accessible via API port 20000.

  The TRACE32 config file for the debug device contains the following lines:

```
…
RCL=NETASSIST
PACKLEN=1024
PORT=20000
…
```

  Alternatively the API Port in **T32Start** has to be configured accordingly for the debug device.



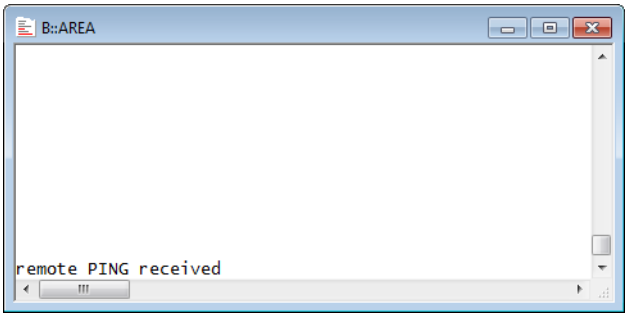

- No error handling is done to keep the script simple.

```
import ctypes   # module for C data types
import enum     # module for enumeration support

# Load TRACE32 Remote API DLL
t32api = ctypes.cdll.LoadLibrary('t32api64.dll')

# TRACE32 Debugger or TRACE32 Instruction Set Simulator as debug device
T32_DEV = 1

# Configure communication channel to the TRACE32 device
# use b for byte encoding of strings
t32api.T32_Config(b"NODE=",b"localhost")
t32api.T32_Config(b"PORT=",b"20000")
t32api.T32_Config(b"PACKLEN=",b"1024")

# Establish communication channel
rc = t32api.T32_Init()
rc = t32api.T32_Attach(T32_DEV)
rc = t32api.T32_Ping()

# TRACE32 control commands

# Release communication channel
rc = t32api.T32_Exit()
```

The Python script is using the following TRACE32 Remote API functions:

```
# configure the communication channel to the TRACE32 device
int T32_Config ( const char *string1, const char *string2 );

# initialize the communication channel
int T32_Init ( void );

# connect to the debug device
int T32_Attach ( int dev );

# ping the debug device
int T32_Ping ( void );

# disconnect from the debug device
int T32_Exit ( void );
```

The following message is displayed in the **TRACE32 Message Area** when the Python script pings the debug device:



The Python script below shows a typical command sequence

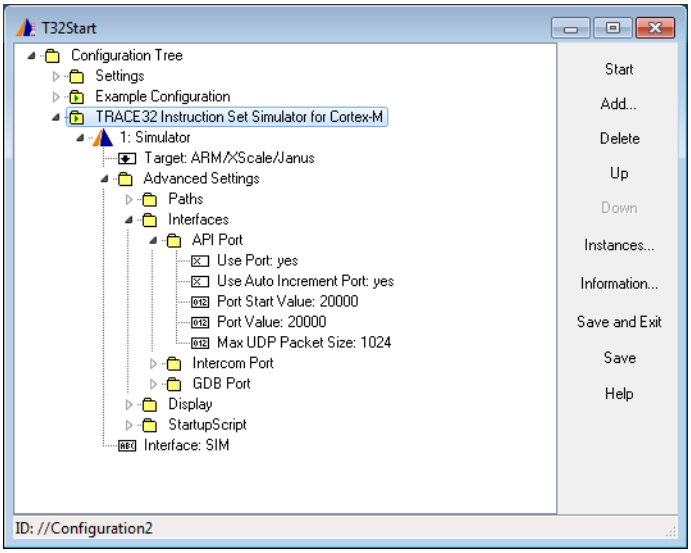
- That establishes the communication between Python and a debug device.
- That releases the communication between Python and a debug device.

The example assumes the following:

- You are working on a 64-bit Windows system.
- You are using a TRACE32 debugger or a TRACE32 Instruction Set Simulator as debug device.
- The TRACE32 config file for the debug device you want to start contains the following lines:

```
...
RCL=NETASSIST
PACKLEN=1024
PORT=20000
...
```

Alternatively the API Port in **T32Start** has to be configured accordingly for the debug device.




- No error handling is done to keep the script simple.

```
import ctypes              # module for C data types
import enum                # module for C data types
import os                  # module for paths and directories
import subprocess          # module to create an additional process
import time                # time module

# TRACE32 Debugger or TRACE32 Instruction Set Simulator
T32_DEV = 1

# Start TRACE32 instance
t32_exe = os.path.join('C:' + os.sep, 'T32_DVD_2_2016',
                       'bin', 'windows64', 't32marm.exe')
config_file = os.path.join('C:' + os.sep, 'T32_DVD_2_2016', 'config.t32')
start_up = os.path.join('C:' + os.sep, 'T32_DVD_2_2016', 'demo', 'arm',
                       'compiler', 'arm', 'cortexm.cmm')

#command = ["C:\T32\bin\windows64\t32marm.exe",
#           '-c', "C:\T32\config.t32",
#           '-s', "C:\T32\demo\arm\compiler\arm\cortexm.cmm"]
command = [t32_exe, '-c', config_file, '-s', start_up]

process = subprocess.Popen(command)

# Wait until the TRACE32 instance is started
time.sleep(5)

# Load TRACE32 Remote API
t32api = ctypes.cdll.LoadLibrary('t32api64.dll')

# Configure communication channel
t32api.T32_Config(b"NODE=",b"localhost")
t32api.T32_Config(b"PORT=",b"20000")
t32api.T32_Config(b"PACKLEN=",b"1024")

# Establish communication channel
rc = t32api.T32_Init()
rc = t32api.T32_Attach(T32Device.T32_DEV_ICD)
rc = t32api.T32_Ping()

# TRACE32 control commands

# Release communication channel
rc = t32api.T32_Exit()
```

## Result as a Message

For the following example the PRACTICE script ends with a **PRINT** *<message>* command. The Python script can read this message and evaluate it as the script´s result.

```
…                                       ; last lines of PRACTICE script
PRINT "Target setup successful"         ; cortexm.cmm
ENDDO
```

```
…
class PracticeInterpreterState(enum.IntEnum):
    UNKNOWN = -1
    NOT_RUNNING = 0
    RUNNING = 1
    DIALOG_OPEN = 2

class MessageLineState(enum.IntEnum):
    ERROR = 2
    ERROR_INFO = 16

# Start PRACTICE script
t32api.T32_Cmd(b"CD.DO ~~/demo/arm/compiler/arm/cortexm.cmm")

# Wait until PRACTICE script is done
state = ctypes.c_int(PracticeInterpreterState.UNKNOWN)
rc = 0
while rc==0 and not state.value==PracticeInterpreterState.NOT_RUNNING:
    rc = t32api.T32_GetPracticeState(ctypes.byref(state))

# Get confirmation that everything worked
status = ctypes.c_uint16(-1)
message = ctypes.create_string_buffer(256)
rc = t32api.T32_GetMessage(ctypes.byref(message), ctypes.byref(status))

if rc == 0
   and not status.value == MessageLineState.ERROR
   and not status.value == MessageLineState.ERROR_INFO:
   print(message.value)


…
```

The script is using the following TRACE32 Remote API functions:

```
# execute a TRACE32 command
int T32_Cmd ( const char *command );
```

**T32_Cmd** is blocking. The TRACE32 Remote API provides the return value after the command execution is completed. There is no time-out.

If you are using the **DO** command to start a PRACTICE script you have to be aware that TRACE32 provides the return value as soon as the script is successfully started!!!

You have to use the following function to check if the processing of the script is completed.

```
# check if started PRACTICE script is still running
# the function returns 0 if no PRACTICE script is running
int T32_GetPracticeState ( int *pstate );
```

The call of the function **t32api.T32_GetPracticeState** illustrates how C compatible data types are used in Python.

```
state = ctypes.c_int(PracticeInterpreterState.UNKNOWN)
rc = 0
while rc==0 and not state.value==PracticeInterpreterState.NOT_RUNNING:
    rc = t32api.T32_GetPracticeState(ctypes.byref(state))
```

Finally you may want to know, if the PRACTICE script was executed without errors. The following command allows you to read the message text printed to the **TRACE32 Message Line**.

```
# get content of the TRACE32 Message Line
int T32_GetMessage ( char message[256], uint16_t *status );
```

The script on the previous page does not contain any error handling. Here an example for an error handling for the following three error types:

| | |
|---|---|
| **error 1** | Communication error with TRACE32 Remote API.<br><br>`t32api.T32_Cmd(b"<command>")` return value < 0. |
| **error 2** | Error in command execution e.g. specified script not found.<br><br>`t32api.T32_Cmd(b"<command>")` return value == 0<br>and<br>`t32api.T32_GetMessage(ctypes.byref(message), ctypes.byref(status))`<br>has set status.value == 2 or 16 |
| **error 3** | Command is unknown or locked e.g. command is unknown due to typo in command name.<br>`t32api.T32_Cmd(b"<command>")` return value > 0. |

```
rc=t32api.T32_Cmd(b"<command>")

# error 1
if rc < 0:
    rc = t32api.T32_Exit()
    raise ConnectionError("TRACE32 Remote API communication error")

else:
    status = ctypes.c_uint16(-1)
    message = ctypes.create_string_buffer(256)
    mrc = t32api.T32_GetMessage(ctypes.byref(message), ctypes.byref(status))
    if mrc != 0:
       rc = t32api.T32_Exit()
       raise ConnectionError("TRACE32 Remote API communication error")
# error 2
    elif rc == 0 and ((status.value == 2) or (status.value == 16)):
        print ("TRACE32 error message: " + message.value.decode("utf-8"))
        t32api.T32_Cmd(b"PRINT")
# error 3
    elif rc > 0:
        print ("TRACE32 error message: " + message.value.decode("utf-8"))
        t32api.T32_Cmd(b"PRINT")
```

Since the function **T32_GetMessage** reads the message text, but does not reset it, you have to send an empty PRINT command to delete the message text.

## Result via EVAL Command

For the following example the PRACTICE script ends with a **EVAL** *<expression>* command. The Python script can read the command result and evaluate it as the script´s result.

```
…                                       ; last lines of PRACTICE script
EVAL 0.                                 ; cortexm.cmm
ENDDO
```

```
…

# Start PRACTICE script
t32api.T32_Cmd(b"CD.DO ~~/demo/arm/compiler/arm/cortexm.cmm")

# Wait until PRACTICE script is done
state = ctypes.c_int(PracticeInterpreterState.UNKNOWN)
rc = 0
while rc == 0
    and not state.value == PracticeInterpreterState.NOT_RUNNING:
    rc = t32api.T32_GetPracticeState(ctypes.byref(state))

# Get confirmation that everything worked
eval = ctypes.c_uint16(-1)
rc = t32api.T32_EvalGet(ctypes.byref(eval))

if rc == 0 and eval.value == 0:
   print("Target setup completed")

…
```

The script is using the following new TRACE32 Remote API functions:

```
# get result of EVAL command
int T32_EvalGet ( uint32_t *pEvalResult );
```

The following two TRACE32 Remote API functions can also be used to work with TRACE32 functions.

```
# execute a TRACE32 command
int T32_Cmd ( const char *command );
```

```
# get result of EVAL command
int T32_EvalGet ( uint32_t *pEvalResult );
```

```
…
rc == t32api.T32_Cmd(b"EVAL hardware.POWERDEBUG()")

eval = ctypes.c_uint16(-1)
rc = t32api.T32_EvalGet(ctypes.byref(eval))
…
```

The TRACE32 function **hardware.POWERDEBUG()** returns true if the connected TRACE32 tool includes a PowerDebug Module.

If the TRACE32 function returns a string the following TRACE32 Remote API function has to be used:

```
# get result of EVAL command if it is a string
int T32_EvalGetString ( char* EvalString );
```

```
…
rc == t32api.T32_Cmd(b"EVAL SOFTWARE.VERSION()")

eval_string = ctypes.create_string_buffer(256)
rc = t32api.T32_EvalGetString(ctypes.byref(eval_string))
…
```

The TRACE32 function **SOFTWARE.VERSION()** returns the current version of the TRACE32 software as a string.

```
…

# Get details for symbol flags[3]
vname = b"flags[3]"
vaddr = ctypes.c_int32(0)
vsize = ctypes.c_int32(0)
vaccess = ctypes.c_int32(0)
rc = t32api.T32_GetSymbol(vname,ctypes.byref(vaddr),ctypes.byref(vsize),
                          ctypes.byref(vaccess))

# Set a write breakpoint to flags[3]
t32api.T32_WriteBreakpoint(vaddr.value,0,16,vsize.value)

# Start program

t32api.T32_Go()

# Wait for breakpoint hit

pstate = ctypes.c_uint16(-1)
while rc == 0 and not pstate.value == 2:
      rc=t32api.T32_GetState(ctypes.byref(pstate))

# Read variable

vvalue = ctypes.c_int32(0)
vvalueh = ctypes.c_int32(0)
rc = t32api.T32_ReadVariableValue(vname,ctypes.byref(vvalue),
                                  ctypes.byref(vvalueh))
print("flags[3]= " + str(vvalue.value))
…
```

The script is using the following TRACE32 Remote API functions:

```
# get details about the specified symbol
int T32_GetSymbol ( const char *symbol,
            uint32_t   *address,
            uint32_t   *size,
            uint32_t   *access );
```

The symbol address and the symbol size is needed to set the breakpoint. The access class can be ignored.

```
# set breakpoint
int T32_WriteBreakpoint ( uint32_t  address,
                int      access,
                int      breakpoint,
                int      size );
```

```
# start the program execution
int T32_Go ( void );
```

```
# check debug state
int T32_GetState ( int *pstate );
```

Debug state is 2 when the program execution is stopped.

```
# read variable
int T32_ReadVariableValue ( const char  *symbol,
                            uint32_t    *value,
                            uint32_t    *hvalue );
```

The example above works if the program execution is stopped after the write access to the variable (break-after-make). If the program execution is stopped just before the write access (break-before-make) a single step has to be performed before the variable value is read.