About this Manual

This manual gives some basic examples that show how to control TRACE32 using a Python 3 script.

The `~/demo/api/python` folder provides some example scripts.

Environment

A TRACE32 debugger can be controlled by Python via the TRACE32 Remote API. The API is built as a plain C source library with a C function interface.
Related Documents

It is recommended to check “API for Remote Control and JTAG Access” (api_remote.pdf) before you read this manual.

Since the TRACE32 Remote API is written in C, function parameters and return value have to be of C compatible data types. Please refer to “ctypes — A foreign function library for Python” for more information.

https://docs.python.org/3/library/ctypes.html#fundamental-data-types
Establish and Release the Communication to the Debug Device

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 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:

```plaintext
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:

```c
# 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()
Run a PRACTICE Script

Result as a Message

For the following example the PRACTICE script ends with a \texttt{PRINT} \texttt{<message>} command. The Python script can read this message and evaluate it as the script’s result.

```
PRINT "Target setup successful"
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:

```python
# 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.

```python
# 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.

```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**.

```python
# 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:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
<th>Code Snippet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Communication error with TRACE32 Remote API.</td>
<td><code>t32api.T32_Cmd(b&quot;&lt;command&gt;&quot;)</code> return value &lt; 0.</td>
</tr>
<tr>
<td>2</td>
<td>Error in command execution e.g. specified script not found.</td>
<td><code>t32api.T32_Cmd(b&quot;&lt;command&gt;&quot;)</code> return value == 0 and <code>t32api.T32_GetMessage(ctypes.byref(message), ctypes.byref(status))</code> has set status.value == 2 or 16</td>
</tr>
<tr>
<td>3</td>
<td>Command is unknown or locked e.g. command is unknown due to typo in command name.</td>
<td><code>t32api.T32_Cmd(b&quot;&lt;command&gt;&quot;)</code> return value &gt; 0.</td>
</tr>
</tbody>
</table>

```python
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
```

```python
...

# 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.

```c
int T32_Cmd ( const char *command );
```

```c
int T32_EvalGet ( uint32_t *pEvalResult );
```

```c
...  
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:

```c
int T32_EvalGetString ( char* EvalString );
```

```c
...  
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.
Monitor a Variable

...  

```python
# Get details for symbol flags[3]
vname = b"flags[3]"
vaddr = ctypes.c_int32(0)
vsiz​e = 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:

```c
# 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.
Debug state is 2 when the program execution is stopped.

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.