Training Script Language PRACTICE

Training Script Language PRACTICE

Version 09-Sep-2015

History

The most current version of this training manual can be downloaded from:

http://www.lauterbach.com/training.html

31-Jul-15	Partly revised.

E-Learning

Videos about the script language PRACTICE can be found here:

http://www.lauterbach.com/tut_practice.html

Ready-to-Run Scripts

Ready-to-run PRACTICE scripts provided by the Lauterbach experts are published and updated daily here:

http://www.lauterbach.com/scripts.html

Introduction to Script Language PRACTICE

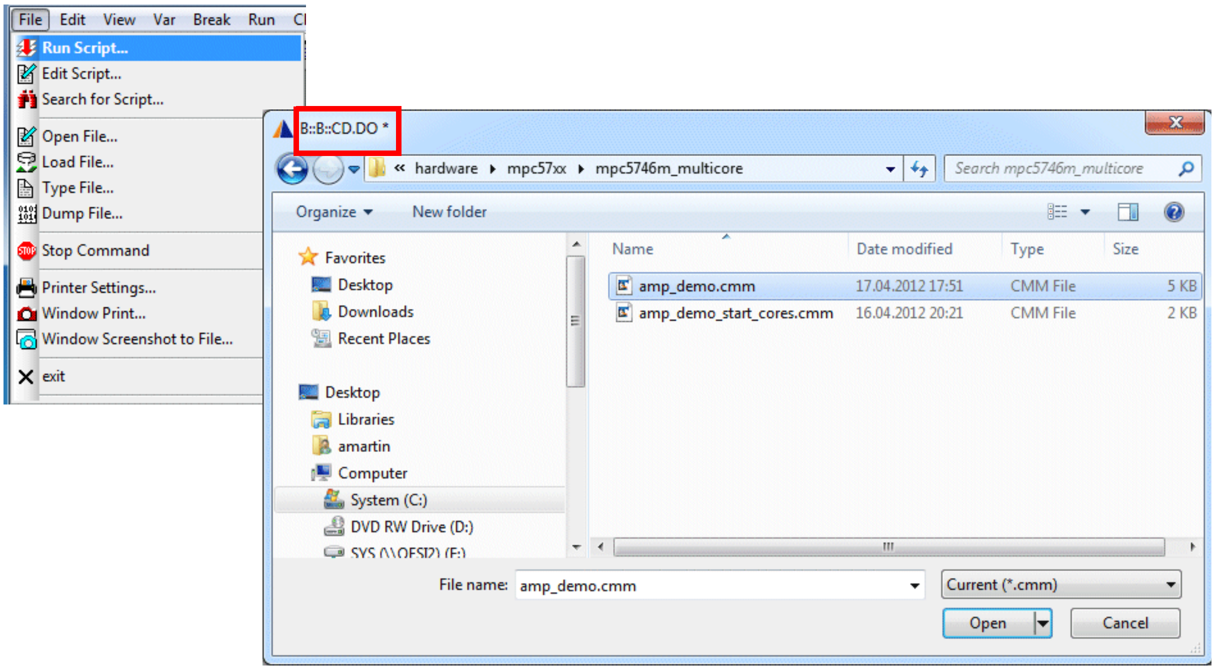
Area of Use

The main tasks of PRACTICE scripts are:

- to provide the proper start-up sequence for the development tool
- to automate FLASH programming
- to customize the user interface
- to store and reactivate specific TRACE32 settings
- to run automatic tests

The standard extension for PRACTICE scripts is **`.cmm`**.

Run a Script



Name	Date modified	Type	Size
amp_demo.cmm	17.04.2012 17:51	CMM File	5 KB
amp_demo_start_cores.cmm	16.04.2012 20:21	CMM File	2 KB

File name: amp_demo.cmm

Current (*.cmm)

```
CD.DO *                         // "*" opens a file browser for script
                                // selection

                                // TRACE32 first changes to the directory
                                // where the selected script is located and
                                // then starts the script
```

ChDir.DO *<filename>*	Change to the directory where the script <*filename*> is located and start the script.
DO *<filename>*	Start script <*filename*>.
PATH *[+] <path_name>*	Define search paths for PRACTICE scripts.

```
DO memtest

ChDir.DO c:/t32/demo/powerpc/hardware

PATH c:/t32/tests
```

Convert TRACE32 Settings to a Script

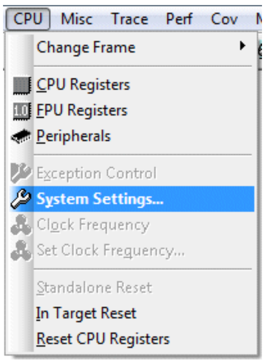
The commands **STOre** and **ClipSTOre** generate scripts that allow to reactivate the specified TRACE32 *<setting>* at any time.

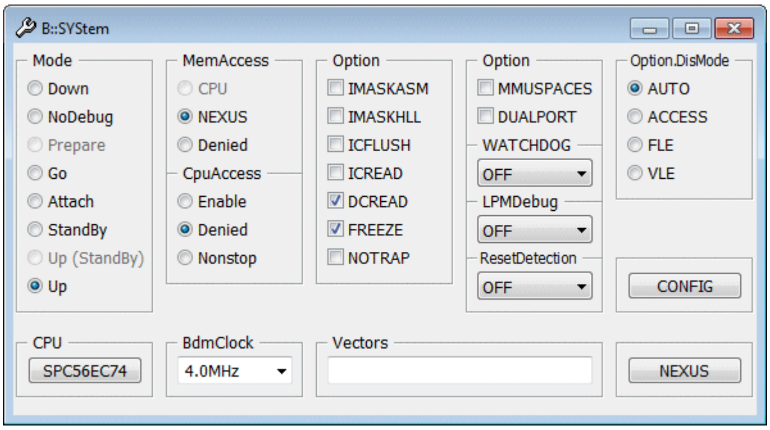
<setting> is in most cases the set-up of a command group.

<setting>	
SYStem	Setting for command group SYStem.
Break	Setting for command group Break.
Win	TRACE32 window configuration.
NoDate	Do not include time stamp into the script.

ClipSTOre {*<setting>*}	TRACE32 creates a script that allows to reactivate the selected settings. The script is stored to the clipboard.
STOre *<filename>* {*<setting>*}	TRACE32 creates the script *<filename>* to reactivate the selected settings.

Create a script to reactivate the current SYStem settings and store it to clipboard




B::SYStem

Mode: Down, NoDebug, Prepare, Go, Attach, StandBy, Up (StandBy), [✓ Up]

MemAccess: CPU, [✓ NEXUS], Denied

CpuAccess: Enable, [✓ Denied], Nonstop

Option: IMASKASM ☐, IMASKHLL ☐, ICFLUSH ☐, ICREAD ☐, DCREAD ☑, FREEZE ☑, NOTRAP ☐

Option: MMUSPACES ☐, DUALPORT ☐

WATCHDOG: OFF

LPMDebug: OFF

ResetDetection: OFF

Option.DisMode: [✓ AUTO], ACCESS, FLE, VLE

CONFIG

CPU: SPC56EC74

BdmClock: 4.0MHz

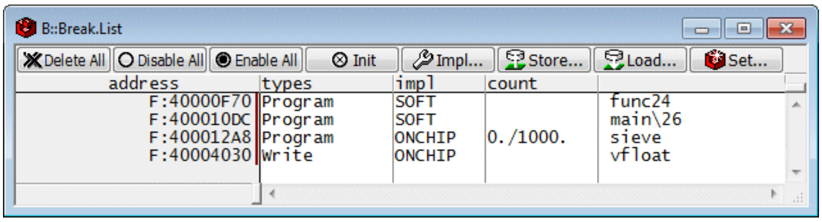
Vectors:

NEXUS

```
ClipSTOre SYStem NoDate
```

```
B::

SYSTEM.RESET
SYSTEM.CPU SPC56EC74
SYSTEM.CONFIG CORENUMBER 2.
SYSTEM.CONFIG CORE 1. 1.
CORE.ASSIGN 1.
SYSTEM.MEMACCESS NEXUS
SYSTEM.CPUACCESS DENIED
SYSTEM.OPTION IMASKASM OFF
SYSTEM.OPTION IMASKHLL OFF
SYSTEM.BDMCLOCK 4000000.
SYSTEM.CONFIG TRISTATE OFF
SYSTEM.CONFIG SLAVE OFF
SYSTEM.CONFIG TAPSTATE  7.
SYSTEM.CONFIG TCKLEVEL  0.
SYSTEM.CONFIG.DEBUGPORT Analyzer0
SYSTEM.CONFIG CJTAGFLAGS  0x3
SYSTEM.MODE UP

ENDDO
```

Create a script to reactivate current Break settings and store it to clipboard



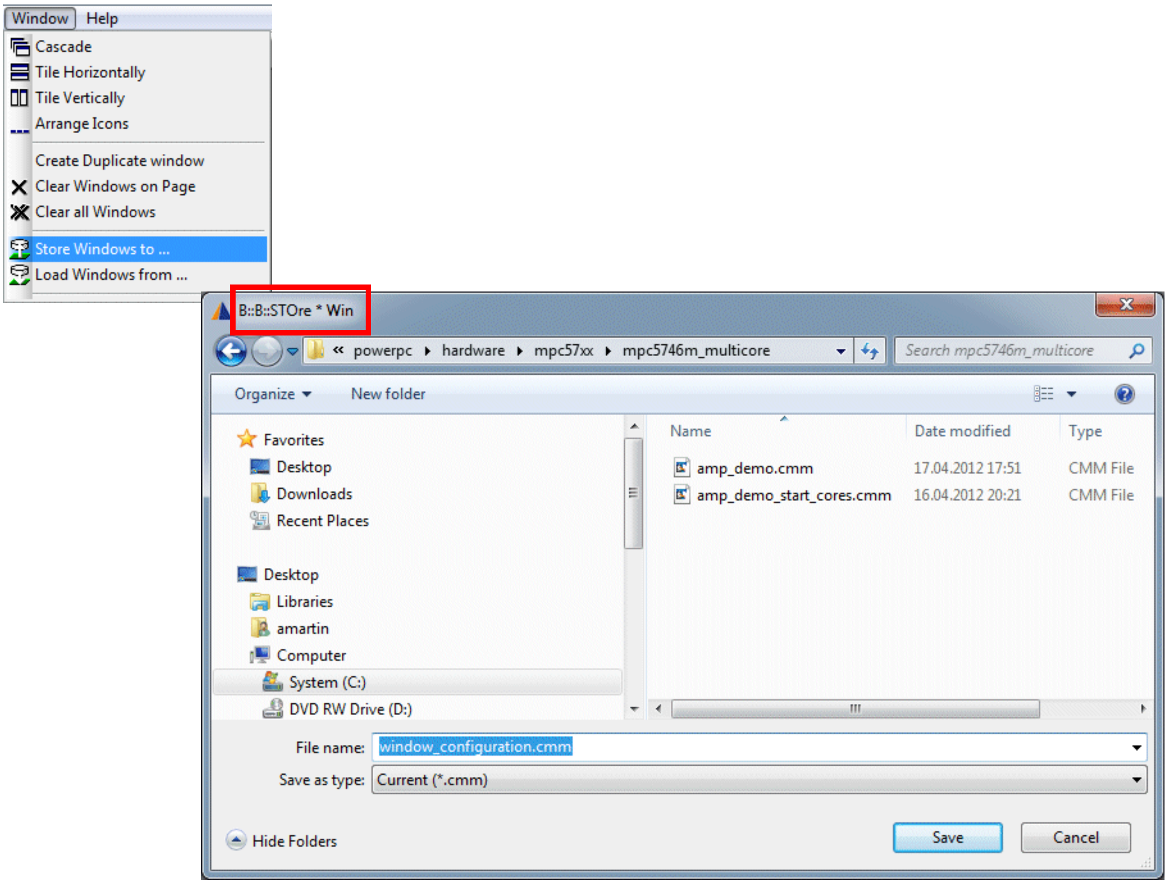
```
            address types      impl       count
        F:40000F70 Program    SOFT                  func24
        F:400010DC Program    SOFT                  main\26
        F:400012A8 Program    ONCHIP     0./1000.   sieve
        F:40004030 Write      ONCHIP                vfloat
```

```
ClipSTOre Break NoDate
```

```
B::

BREAK.RESET
B.S   func24 /P
B.S   main\26 /P
B.S   sieve /P /O /COUNT 1000.
V.B.S vfloat; /W

ENDDO
```

The breakpoints are saved at a symbolic level by default.

Create a script to reactivate the current window configuration



Name	Date modified	Type
amp_demo.cmm	17.04.2012 17:51	CMM File
amp_demo_start_cores.cmm	16.04.2012 20:21	CMM File

File name: window_configuration.cmm

Save as type: Current (*.cmm)

```
STOre window_configuration.cmm Win NoDate       // TRACE32 creates script
                                                // window_configuration
                                                // to reactivate the
                                                // current window
                                                // configuration
```

```
B::

TOOLBAR ON
STATUSBAR ON
FramePOS 15.625 8.9286 193. 47.
WinPAGE.RESet

WinPAGE.Create P000
WinCLEAR

WinPOS 0.0 22.214 80. 5. 0. 0. W002
Var.View %SpotLight.on %E flags %Open vtripplearray

WinPOS 0.0 31.429 80. 8. 5. 0. W003
Frame /Locals /Caller

WinPOS 0.0 0.0 80. 16. 13. 1. W000
WinTABS 10. 10. 25. 62.
List.auto

WinPOS 84.25 0.0 77. 20. 0. 0. W004
Register.view

WinPOS 83.875 24.071 105. 6. 0. 0. W001
PER , "FlexCAN"

WinPAGE.select P000

ENDDO
```

Command LOG

The LOG command allows to record most of the activities in the TRACE32 PowerView GUI.

Commands to control the command LOG:

LOG.OPEN *<file>*	Create and open a file for the command LOG. The default extension for LOG files is (**`.log`**).
LOG.CLOSE	Close the command LOG file.
LOG.OFF	Switch off command LOG temporarily.
LOG.ON	Switch on command LOG.
LOG.type	Display command LOG while recording.

```
LOG.OPEN my_log         Creates and opens the .log file

TYPE my_log.log         Displays .log file contents while recording

…                       Recording

LOG.CLOSE               Closes .log file
```

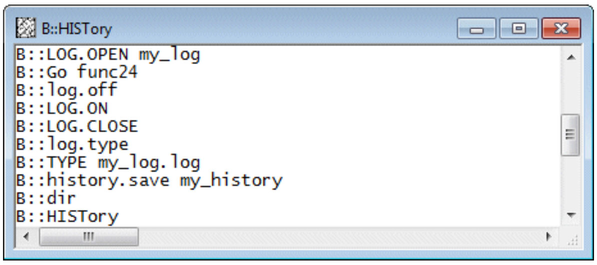
Contents of a command-LOG

```
 B::B::List
 B::Go func24
// B::LOG.ON
 B::B::PER , "Analog to Digital Converter"
 B::B::PER.Set.simple ANC:0xFFE00000 %L (d.l(ANC:0xFFE00000)&~0x40000000)|0x40000000
```

Command History

!	The command history records only commands entered into the command line. The default extension for HISTory-files is (**`.log`**)

HISTory.type Display the command history



```
B::LOG.OPEN my_log
B::Go func24
B::log.off
B::LOG.ON
B::LOG.CLOSE
B::log.type
B::TYPE my_log.log
B::history.save my_history
B::dir
B::HISTory
```

HISTory.SAVE [<*filename*>]	Save the command history
HISTory.SIZE [<*size*>]	Define the size of the command history

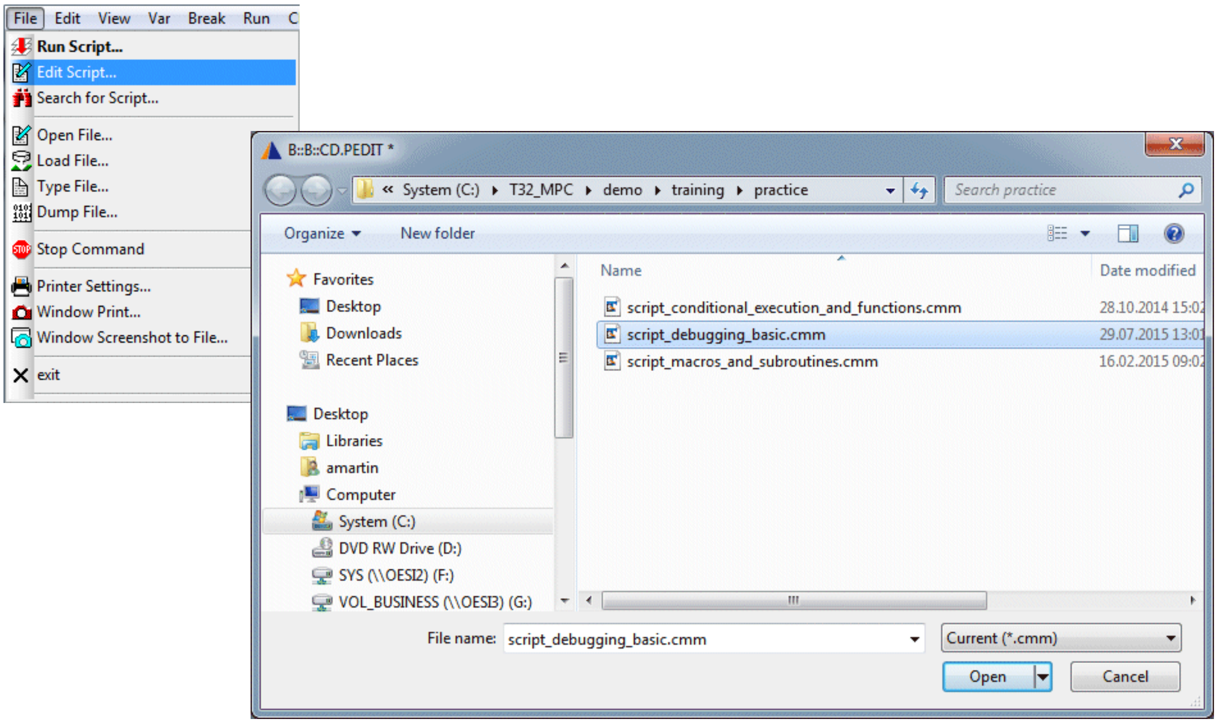
If the file **`T32.cmm`** contains the instruction:

```
AutoSTOre , HISTory
```

then the command history is automatically saved in the TMP directory at the exit of TRACE32 and recalled when TRACE32 is started.

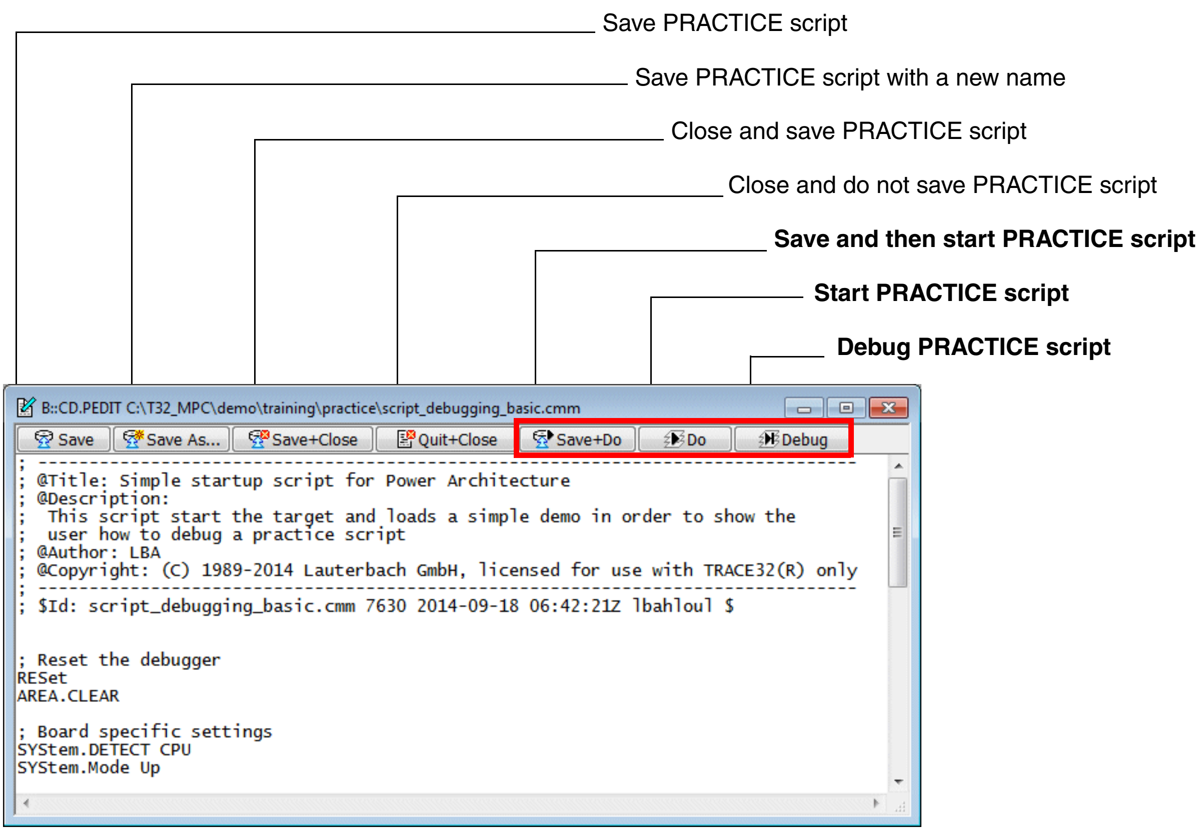
AutoSTOre <*filename*> {<*setting*>}	Store defined settings automatically at the exit of TRACE32 and reactivate them at the start of TRACE32

Script Editor PEDIT



```
CD.PEDIT *                   // "*" opens a file browser for script
                             // selection

                             // TRACE32 first changes to the directory
                             // where the selected script is located and
                             // then opens the script in a PEDIT window
```

ChDir.PEDIT <*filename*>	Change to the directory where the script <*filename*> is located and open script in script editor PEDIT.
PEDIT <*filename*>	Open script <*filename*> in script editor PEDIT.


Save PRACTICE script
Save PRACTICE script with a new name
Close and save PRACTICE script
Close and do not save PRACTICE script
Save and then start PRACTICE script
Start PRACTICE script
Debug PRACTICE script
B::CD.PEDIT C:\T32_MPC\demo\training\practice\script_debugging_basic.cmm
Save
Save As...
Save+Close
Quit+Close
Save+Do
Do
Debug

```
; --------------------------------------------------------------------------------
; @Title: Simple startup script for Power Architecture
; @Description:
;  This script start the target and loads a simple demo in order to show the
;  user how to debug a practice script
; @Author: LBA
; @Copyright: (C) 1989-2014 Lauterbach GmbH, licensed for use with TRACE32(R) only
; --------------------------------------------------------------------------------
; $Id: script_debugging_basic.cmm 7630 2014-09-18 06:42:21Z lbahloul $


; Reset the debugger
RESet
AREA.CLEAR

; Board specific settings
SYStem.DETECT CPU
SYStem.Mode Up
```

Syntax Highlighting

The script editor PEDIT, unfortunately does not support syntax highlighting for PRACTICE scripts.

If you want to have syntax highlighting for PRACTICE scripts, you have to use the following workaround.

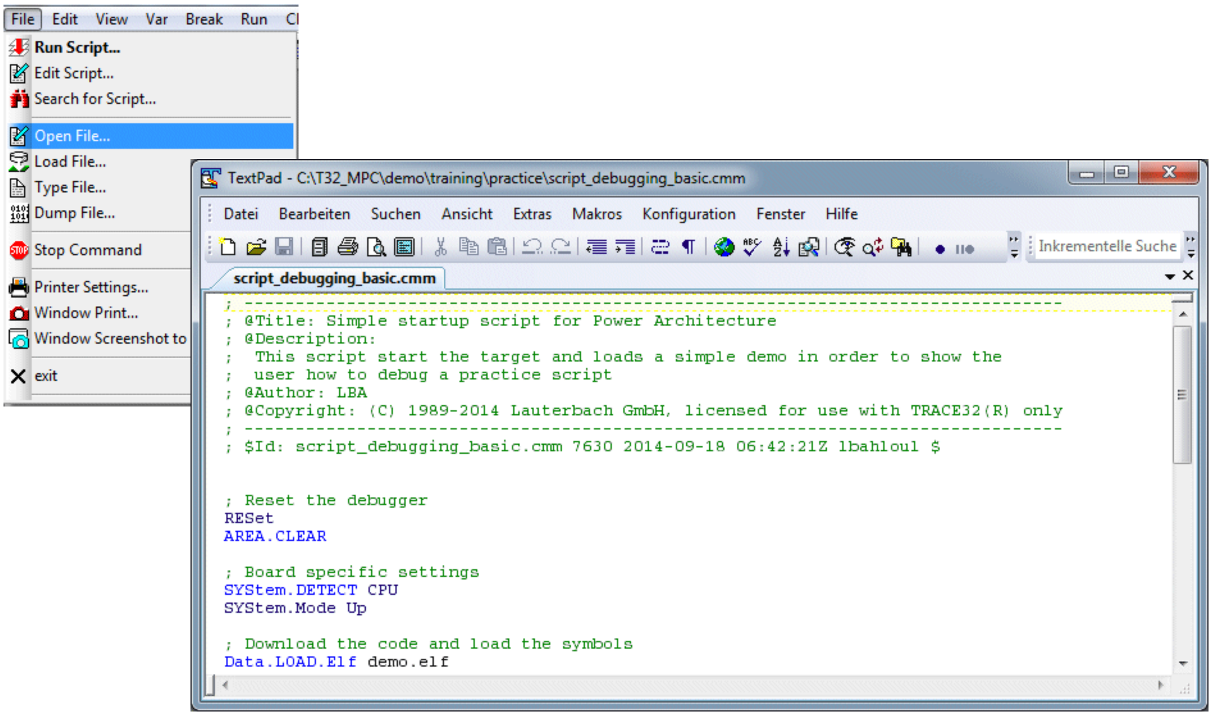
1. Redirect the call of the TRACE32 editor EDIT to an external editor by using the command TRACE32 command **SETUP.EDITEXT**.

2. Install syntax highlighting files provided by Lauterbach for the external editor.

EDIT *<filename>*	Open file with standard TRACE32 editor.
SETUP.EDITEXT ON *<command>*	Advise TRACE32 to use the specified external editor if the EDIT command is used. *<command>* contains the command that TRACE32 sends to your host OS to start the external editor. In this string the following replacements will be made: • **`*`** will be replaced by the actual file name. • **#** will be replaced by the actual line number.

Lauterbach provides syntax highlighting files for some common text editors. Please refer to **`~~\demo\practice\syntaxhighlighting`** for details. `~~` stands for the `<TRACE32_installation_directory>`, which is `c:/T32` by default.

```
// Advise TRACE32 to use TextPad when the EDIT command is used
SETUP.EDITEXT ON "C:\Program Files (x86)\TextPad 5\TextPad.exe ""* (#)"""
```



```
; --------------------------------------------------------------------------------
; @Title: Simple startup script for Power Architecture
; @Description:
;   This script start the target and loads a simple demo in order to show the
;   user how to debug a practice script
; @Author: LBA
; @Copyright: (C) 1989-2014 Lauterbach GmbH, licensed for use with TRACE32(R) only
; --------------------------------------------------------------------------------
; $Id: script_debugging_basic.cmm 7630 2014-09-18 06:42:21Z lbahloul $


; Reset the debugger
RESet
AREA.CLEAR

; Board specific settings
SYStem.DETECT CPU
SYStem.Mode Up

; Download the code and load the symbols
Data.LOAD.Elf demo.elf
```

Debugging of PRACTICE Script

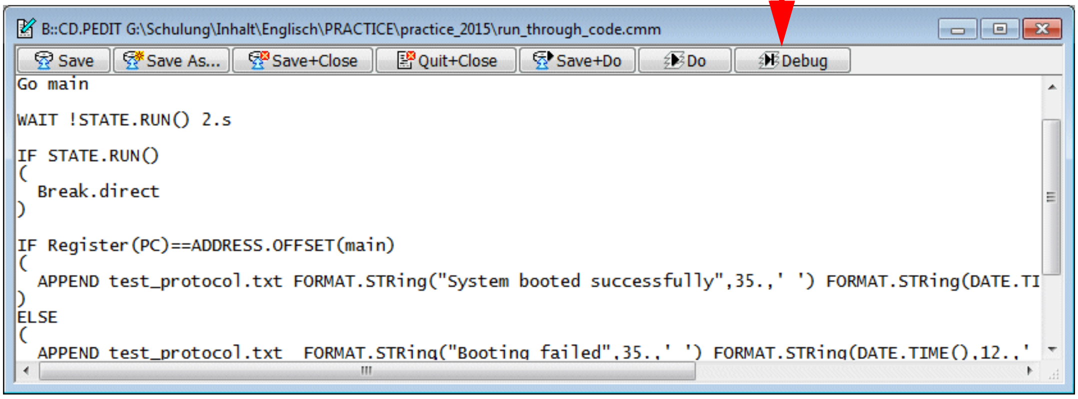
TRACE32 supports the debugging of PRACTICE programs.

A short video that provide an introduction into PRACTICE debugging is available on:

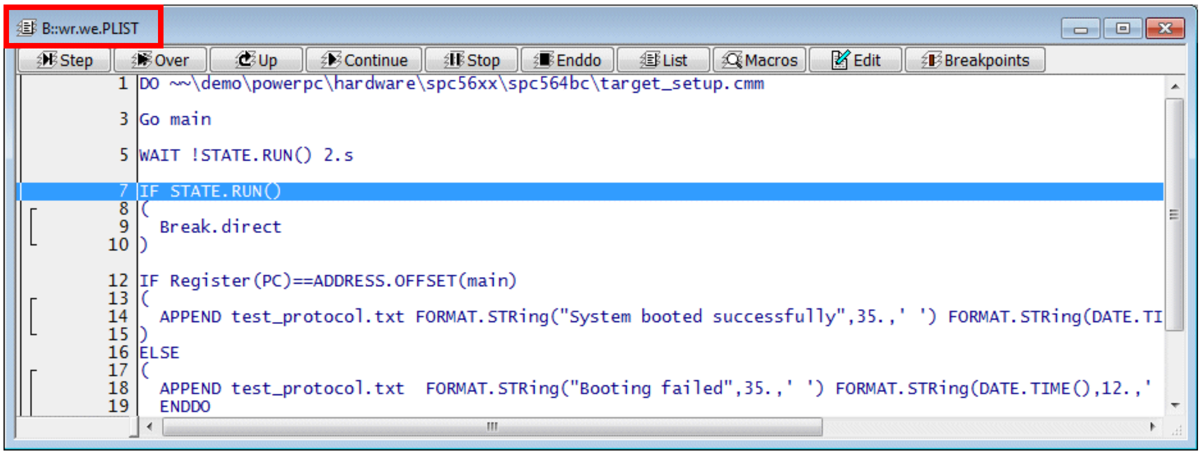
http://www.lauterbach.com/tut_practice.html

Debug Environment

Use ***Debug*** to start the debugger for the PRACTICE script


B::CD.PEDIT G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015\run_through_code.cmm
Save
Save As...
Save+Close
Quit+Close
Save+Do
Do
Debug

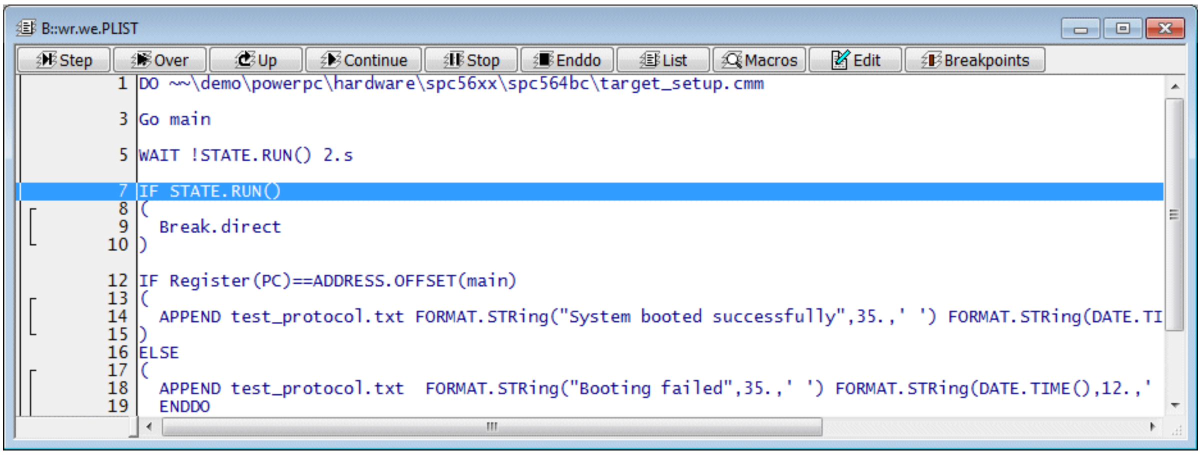
```
Go main

WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)

IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND test_protocol.txt FORMAT.STRing("System booted successfully",35.,' ') FORMAT.STRing(DATE.TI
)
ELSE
(
  APPEND test_protocol.txt  FORMAT.STRing("Booting failed",35.,' ') FORMAT.STRing(DATE.TIME(),12.,'
```



```
DO ~~\demo\powerpc\hardware\spc56xx\spc564bc\target_setup.cmm

Go main

WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)

IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND test_protocol.txt FORMAT.STRing("System booted successfully",35.,' ') FORMAT.STRing(DATE.TI
)
ELSE
(
  APPEND test_protocol.txt  FORMAT.STRing("Booting failed",35.,' ') FORMAT.STRing(DATE.TIME(),12.,'
  ENDDO
```

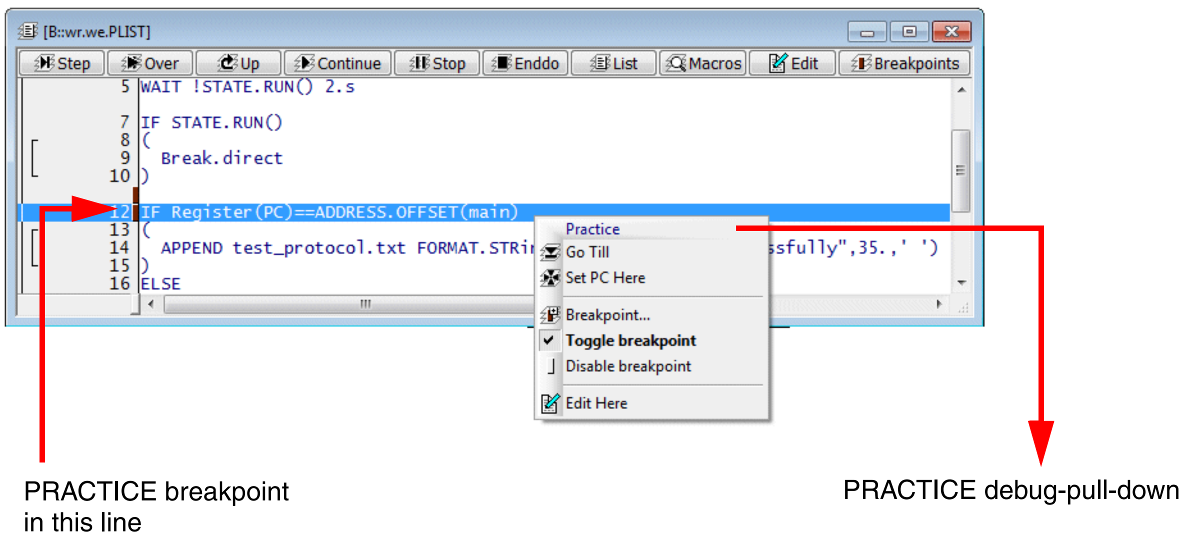
WinResist.WinExt.ChDir.PLIST	**WinResist**: PEDIT window is not deleted by command **WinCLEAR.** **WinExit**: Detach PEDIT window from the TRACE32 main window - even if TRACE32 is operating in MDI window mode.

PSTEP <*filename*>	Start script in PRACTICE debugger.
ChDir.PSTEP <*filename*>	TRACE32 first changes to the directory where the script is located and then starts the script in the PRACTICE debugger.
WinResist.WinExt.ChDir.PSTEP <*filename*>	



```
DO ~~\demo\powerpc\hardware\spc56xx\spc564bc\target_setup.cmm

Go main

WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)

IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND test_protocol.txt FORMAT.STRing("System booted successfully",35.,' ') FORMAT.STRing(DATE.TI
)
ELSE
(
  APPEND test_protocol.txt  FORMAT.STRing("Booting failed",35.,' ') FORMAT.STRing(DATE.TIME(),12.,'
  ENDDO
```

Local buttons in PLIST/PSTEP window	
Step	Single step PRACTICE script (command **PSTEP**).
Over	Run called PRACTICE script or PRACTICE subroutine as a whole (command **PSTEPOVER**).
Up	End current PRACTICE script or subroutine and return to the caller (command **PSTEPOUT**).
Continue	Continue the execution of PRACTICE script (command **CONTinue**).
Stop	Stop the execution of the PRACTICE script (command **STOP**).
Enddo	End the current PRACTICE script. Execution is continued in the calling PRACTICE script. If no calling script exists, the PRACTICE script execution is ended (command **ENDDO**).
List	Open a new PRACTICE debug window (command **WinResist.WinExt.PLIST**).
Macros	Display the PRACTICE stack (command **PMACRO.list**).
Edit	Open PRACTICE editor PEDIT to edit the PRACTICE script (command **WinResist.WinExt.PEDIT**).
Breakpoints	Open a **PBREAK.List** window to display all PRACTICE breakpoints.

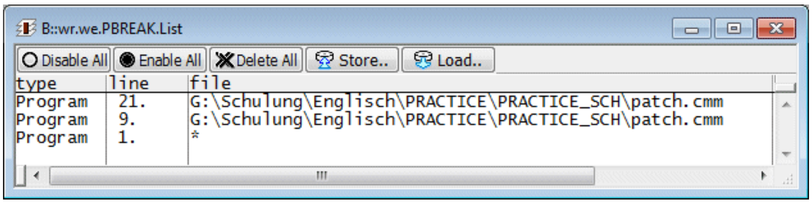


```
WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)

IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND test_protocol.txt FORMAT.STRi                    ssfully",35.,' ')
)
ELSE
```

PRACTICE debug-pull-down	
Goto Till	Run PRACTICE script until the selected line (command **CONTinue** <line_number>).
Set PC Here	Set PRACTICE PC to the selected line.
Breakpoint …	Open **PBREAK.Set** dialog to configure PRACTICE breakpoint.
Toggle breakpoint	Toggle PRACTICE breakpoint.
Disable breakpoint	Disable PRACTICE breakpoint (command **PBREAK.DISable**).
Edit Here	Open PRACTICE editor to edit the PRACTICE script. The cursor is automatically set to the selected line.

PRACTICE breakpoints can be set:

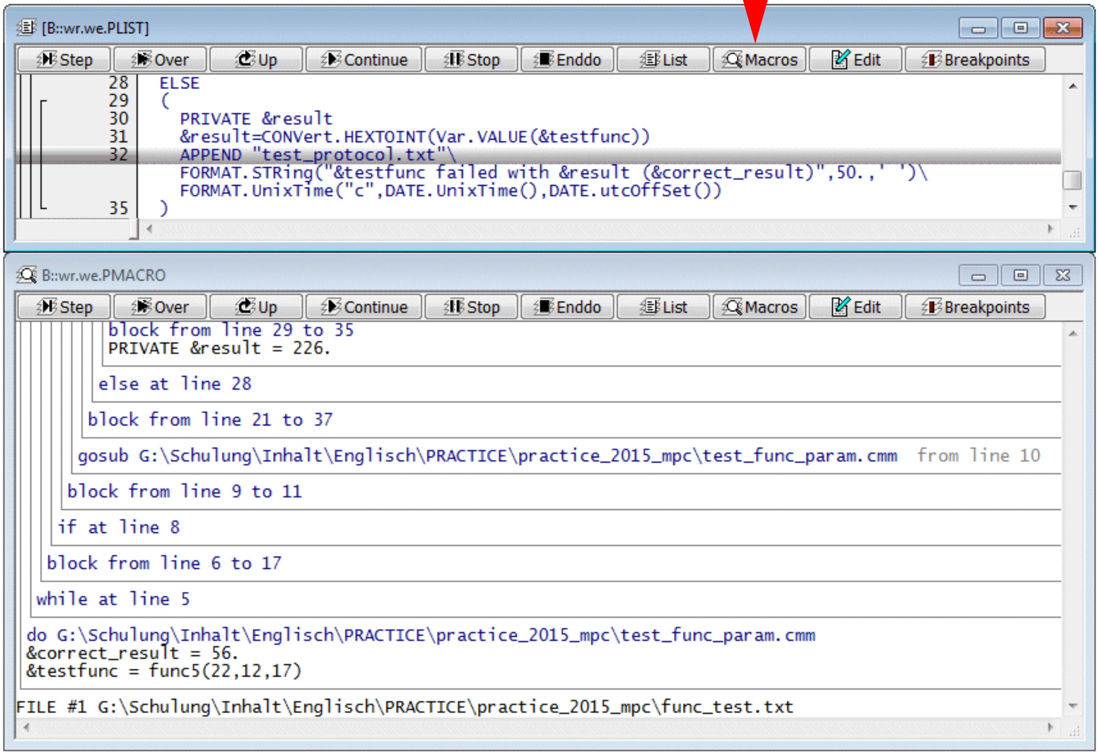
- to a specific line of a specified script
- to a specific line in any script (*)



```
type     line    file
Program   21.    G:\Schulung\Englisch\PRACTICE\PRACTICE_SCH\patch.cmm
Program   9.     G:\Schulung\Englisch\PRACTICE\PRACTICE_SCH\patch.cmm
Program   1.     *
```

Display the PRACTICE Stack

Display the PRACTICE-Stack



```
ELSE
(
  PRIVATE &result
  &result=CONVert.HEXTOINT(Var.VALUE(&testfunc))
  APPEND "test_protocol.txt"\
  FORMAT.STRing("&testfunc failed with &result (&correct_result)",50.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)
```

```
block from line 29 to 35
PRIVATE &result = 226.
else at line 28
block from line 21 to 37
gosub G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_func_param.cmm  from line 10
block from line 9 to 11
if at line 8
block from line 6 to 17
while at line 5
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_func_param.cmm
&correct_result = 56.
&testfunc = func5(22,12,17)
FILE #1 G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\func_test.txt
```

The PRACTICE stack displays the program nesting and the PRACTICE macros.

First Examples

Run Through Program and Generate a Test Report

Task of part 1 of the script: Start the program execution and wait until the program execution is stopped at the entry of the function main.

```
// Script run_through_code.cmm

// Part 1

// Prepare debugging
DO "target_setup.cmm"

Go main
WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break
)
…
```

The script consists of:

TRACE32 commands

Go *<address>*	Start the program execution. Program execution should stop when *<address>* is reached.
Break	Stop the program execution.

PRACTICE commands

DO *<filename>*	Call PRACTICE script *<filename>*
WAIT *<condition> <time_period>*	Wait until *<condition>* becomes true or *<time_period>* expired.

```
IF <condition>
(
  <if_block>
)
ELSE
(
  <else_block>
)
```

Execute *<if_block>* when *<condition>* is true.
Execute *<else_block>* when *<condition>* is false.

PRACTICE is whitespace sensitive. There must be at least one space after a PRACTICE command word.

<block> has to be set in round brackets. PRACTICE requires that round brackets are typed in a separate line.

TRACE32 function

STATE.RUN()	Returns TRUE when the program execution is running. Returns FALSE when the program execution is stopped.

Task of part 2 of the script: Check if program execution stopped at entry to function main and generate a test report.

```
// Part 2
…
IF Register(PC)==ADDRESS.OFFSET(main)
(
 APPEND "test_protocol.txt" FORMAT.STRing("System booted successfully",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  ENDDO
)
```

The backslash \ in conjunction with at least one space serve as a line continuation character.

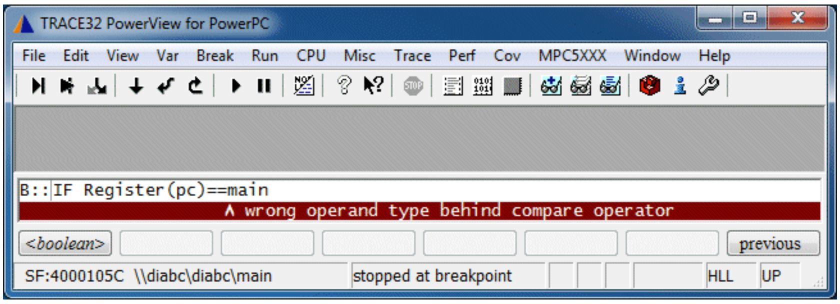
TRACE32 function

// Returns the contents of the specified core register as a hex. number.
Register(*<register_name>*)

Addresses in TRACE32

Why is the following function needed?

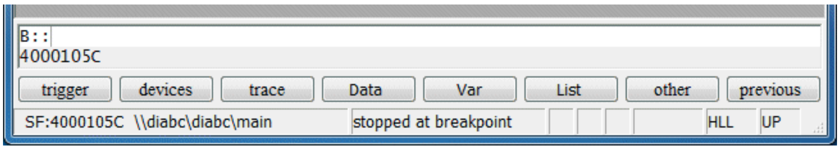
```
ADDRESS.OFFSET(main)
```


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX Window Help
<boolean>
previous

```
B::IF Register(pc)==main
        ^ wrong operand type behind compare operator
```

SF:4000105C \\diabc\diabc\main   stopped at breakpoint   HLL   UP

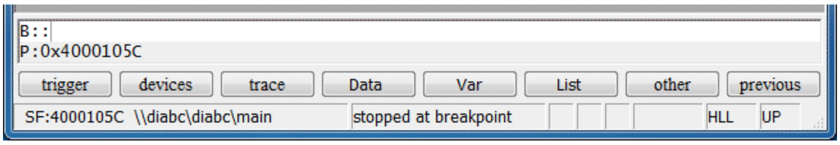
```
PRINT Register(PC)                        // print the content of the program
                                          // counter as a hex. number
```



```
B::
4000105C
```

```
SF:4000105C  \\diabc\diabc\main    stopped at breakpoint    HLL  UP
```

```
PRINT main                                // print the address of main
```



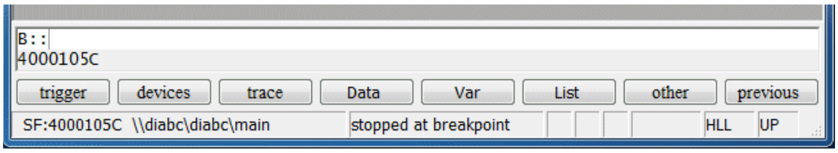
```
B::
P:0x4000105C
```

```
SF:4000105C  \\diabc\diabc\main    stopped at breakpoint    HLL  UP
```

main is an address and addresses in TRACE32 PowerView consist of:

- An **access class** (here P:) which consists of one or more letters/numbers followed by a colon (:)
- A hex. number (here 0x4000105C) that determines the actual address

```
PRINT ADDRESS.OFFSET(main)
```



```
B::
4000105C
```

```
SF:4000105C  \\diabc\diabc\main    stopped at breakpoint    HLL  UP
```

ADDRESS.OFFSET(*<symbol>*)	Returns the hex. number part of an address.

```
// Part 2
…
IF Register(PC)==ADDRESS.OFFSET(main)
(
 APPEND "test_protocol.txt" FORMAT.STRing("System booted successfully",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  ENDDO
)
```

PRATICE commands

APPEND *<filename>* {*<data>*}	Append data to content of file *<filename>.*
ENDDO	A script ends with its last command or with the ENDDO command.

TRACE32 functions

// Formats *<string>* to the specified *<width>.* If the length of *<string>* is shorter the *<width>*,
// the *<filling_character>* is appended.
FORMAT.STRing(*<string>,<width>,<filling_character>***)**

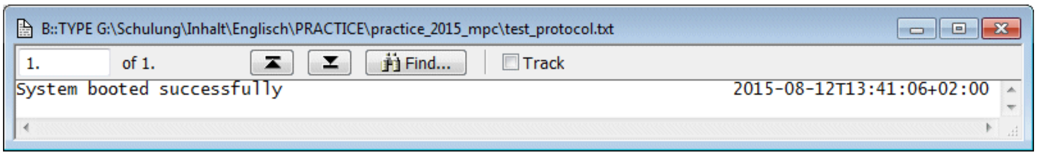
//Returns the time in UNIX format and that is seconds passed since Jan 1st 1970.
DATE.UnixTime()

// Returns offset of the local time to UTC in seconds.
DATE.utcOffset()

// Format Unix time according to ISO 8601
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffset())

```
// Prepare debugging
DO "target_setup.cmm"

Go main

WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)
IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND "test_protocol.txt" FORMAT.STRing("System booted successfully",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  ENDDO
)
…
```

Results for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt
1.
of 1.
Find...
Track

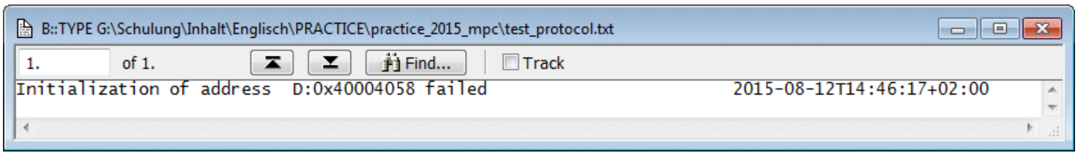
```
System booted successfully                                      2015-08-12T13:41:06+02:00
```

Check Contents of Addresses

Task of the script: After an appropriate program address e.g. main is reached, you can check if certain memory addresses are initialized with their correct value.

```
// Script check_memory_locations.cmm
…
IF Data.Long(D:0x40004058)!=0x0
(
  APPEND "test_protocol.txt"\
  FORMAT.STRing("Initialization of address  D:0x40004058 failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)

IF Data.Long(ANC:0xC3FDC0C4)!=0x0
(
  APPEND "test_protocol.txt"\
  FORMAT.STRing("Initialization of Global Status Register failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
)
…
```

Results for example in


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt
1. of 1.
Find...
Track

```
Initialization of address  D:0x40004058 failed                  2015-08-12T14:46:17+02:00
```

TRACE32 function

Data.Long(*<address>*)	Returns the contents of the specified address as an 32-bit hex. value.

<address> requires an **access class**.

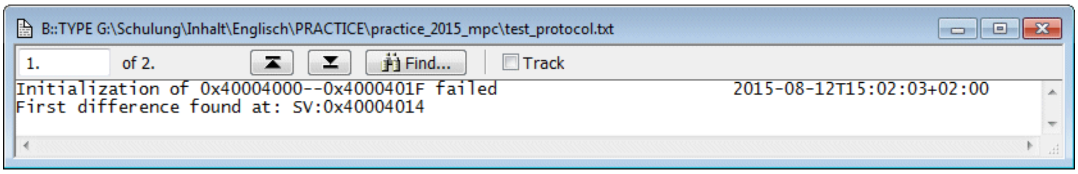
```
Data.Long(D:0x40004058)             // D: indicates the generic access
                                    // class Data

Data.LONG(ANC:0xC3FDC0C4)           // ANC: indicates
                                    // physical address (A)
                                    // No Cache (NC)
```

Check Contents of Address Range

Task of the script: After an appropriate program address e.g. main is reached, you can check if a certain memory range is initialized with their correct values. An easy way to provide the correct values is a binary file.

```
// Script check_memory_range.cmm
…
Data.LOAD.Binary "range_correct" 0x40004000 /DIFF
IF FOUND()
(
  PRIVATE &s
  APPEND "test_protocol.txt"\
  FORMAT.STRing("Initialization of 0x40004000--0x4000401F failed ",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())

  &s=TRACK.ADDRESS()
  APPEND "test_protocol.txt"\
  FORMAT.STRing("First difference found at: &s",70.,' ')
)
…
```

Results for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt
1.
of 2.
Find...
Track

```
Initialization of 0x40004000--0x4000401F failed                2015-08-12T15:02:03+02:00
First difference found at: SV:0x40004014
```

TRACE32 command

Data.LOAD.Binary *<filename> <address>* **/DIFF**	Compare memory content at *<address>* with contents of *<filename>* and provide the result by the following TRACE32 functions:
	FOUND()
	TRACK.ADDRESS()

TRACE32 functions

FOUND()	Returns TRUE if a difference was found.
TRACK.ADDRESS()	Returns the address of the first difference.

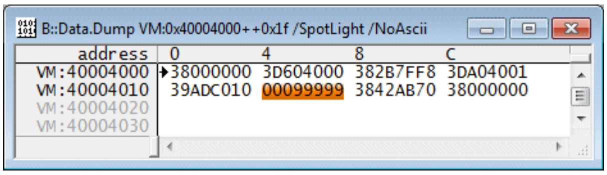
PRACTICE command

PRIVATE {<*macro*>}	Creates a private PRACTICE macro. PRACTICE macros start with **&** to make them different from variables from the program under debug. Private PRACTICE macros are only visible inside the declaring block and are erased when the block ends. In this example the declaring block are the instructions set in round brackets.

To inspect all differences in detail the following script can be helpful.

```
// Script check_memory_range_details.cmm

Data.LOAD.Binary "range_correct" VM:0x40004000
Data.Dump VM:0x40004000++0x1f /SpotLight
SCREEN.display
Data.COPY 0x40004000++0x1f VM:0x40004000
```

Results for example in:



B::Data.Dump VM:0x40004000++0x1f /SpotLight /NoAscii

```
     address  0        4        8        C
 VM:40004000 ▸38000000 3D604000 382B7FF8 3DA04001
 VM:40004010  39ADC010 00099999 3842AB70 38000000
 VM:40004020
 VM:40004030
```

TRACE32 commands

Data.LOAD.Binary *<filename>* **VM:***<address>*	Load the contents of *<filename>* to *<address>* in the **TRACE32 virtual memory**. The TRACE32 Virtual Memory is memory on the host computer which can be displayed and modified with the same commands as real target memory.
Data.dump **VM:***<address_range>* **/SpotLight**	Display the contents of the TRACE32 Virtual Memory for the specified *<address_range>.* The option **SpotLight** advises TRACE32 to mark changed memory locations if the window is displayed in TRACE32 PowerView.
Data.COPY *<address_range>* **VM:***<address>*	Copy the content of *<address_range>* to the TRACE32 Virtual Memory.

PRACTICE commands

If PRACTICE scripts are executed, the screen is only updated after a PRINT command.

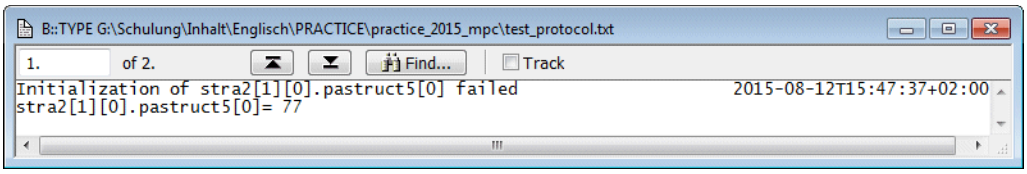
SCREEN.display	Advise TRACE32 to update the screen now.

Check the Contents of Variables

Task of the script: After an appropriate program address e.g. main is reached, you can check if certain variables are initialized with their correct value.

```
// Script check_var.cmm
…
Var.IF stra2[1][0].pastruct5[0]!=0.
(
  APPEND "test_protocol.txt"\
  FORMAT.STRing("Initialization of stra2[1][0].pastruct5[0] failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())

  APPEND "test_protocol.txt" "stra2[1][0].pastruct5[0]= "\
  %Decimal Var.Value(stra2[1][0].pastruct5[0])
)
…
```

My result for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt
1.
of 2.
Find...
Track

```
Initialization of stra2[1][0].pastruct5[0] failed                 2015-08-12T15:47:37+02:00
stra2[1][0].pastruct5[0]= 77
```

PRACTICE command

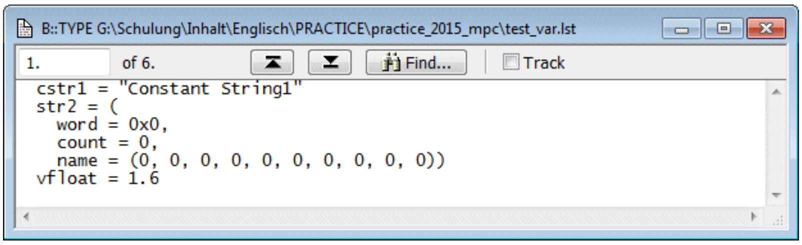
Var.IF *<hll_condition>* **(** *<block>* **)**	Execute *<block>* when the condition written in the programming language used is true (C, C++, …)

PRACTICE function

Var.VALUE(*<hll_expression>*)	Returns the contents of the variable/variable component specified by *<hll_expression>* as a hex. number.

Task of script: Write the content of various variables to a file. Use the same formatting as **Var.View** command.

```
// Script record_var.cmm
…
PRinTer.FileType ASCIIE
PRinTer.OPEN "test_var.lst"

WinPos ,,,,,0
WinPrint.Var.View %String cstr1

WinPos ,,,,,0
WinPrint.Var.View %Open str2

WinPos ,,,,,0
WinPrint.Var.View vfloat

PRinTer.CLOSE
…
```


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_var.lst
1.
of 6.
Find...
Track

```
cstr1 = "Constant String1"
str2 = (
  word = 0x0,
  count = 0,
  name = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
vfloat = 1.6
```

TRACE32 commands

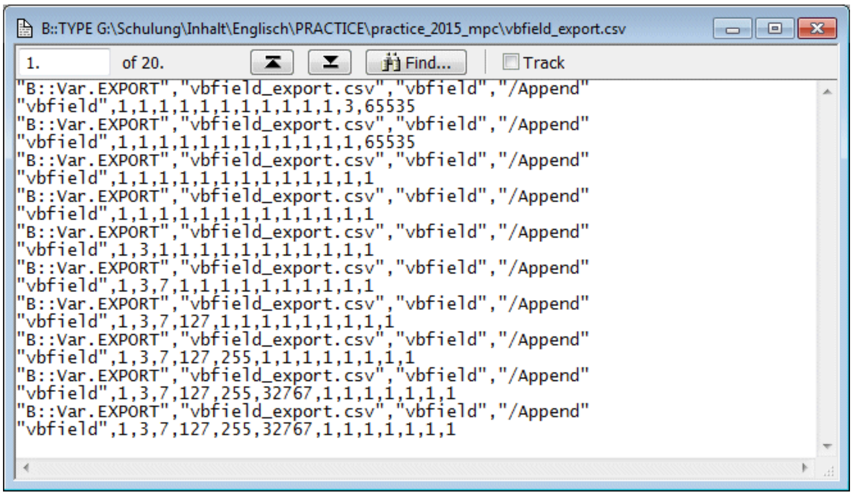
PRinTer.FileType *<format>*	Specify output *<format>* for output file.
PRinTer.Open *<filename>*	Open file *<filename>* for outputs.
PRinTer.CLOSE	Close open output file.
WinPrint.*<command>*	Redirect the *<command>* output to the specified file.
WinPos ,,,,,0	By the default the TRACE32 *<command>* and its output is redirected to the specified file. With this special WINPOS command only the *<command>* output is redirected to the specified file.

Record Variable as CSV

Task of the script: Write the contents of the variable vbfield to a file whenever the program execution stops at the specified breakpoint. Use CSV as output format.

```
// Script test_var_vbfield.cmm
…
Break.RESet
Var.Break.Set vbfield /Write

REPEAT 10.
(
  Go
  WAIT !STATE.RUN() 2.s
  IF STATE.RUN()
  (
     Break
     ENDDO
  )
  Var.EXPORT "vbfield_export.csv" vbfield /Append
)
…
```

Results for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\vbfield_export.csv
1.
of 20.
Find...
Track

```
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,1,1,1,1,1,1,1,1,1,1,3,65535
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,1,1,1,1,1,1,1,1,1,1,1,65535
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,1,1,1,1,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,1,1,1,1,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,1,1,1,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,7,1,1,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,7,127,1,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,7,127,255,1,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,7,127,255,32767,1,1,1,1,1,1,1
"B::Var.EXPORT","vbfield_export.csv","vbfield","/Append"
"vbfield",1,3,7,127,255,32767,1,1,1,1,1,1,1
```

TRACE32 commands

```
// Use expression of your programming language (C, C++, …) to specify write breakpoint
Var.Break.Set <hll_expression> /Write

// Since the number of write breakpoints is limited, it is recommended to reset the current breakpoint
// settings
Break.RESet

// Append content of variables as CSV (Comma Separated Values) to file <filename>
Var.EXPORT <filename> [{%<format>}] {<variable>} /Append
```

PRACTICE command

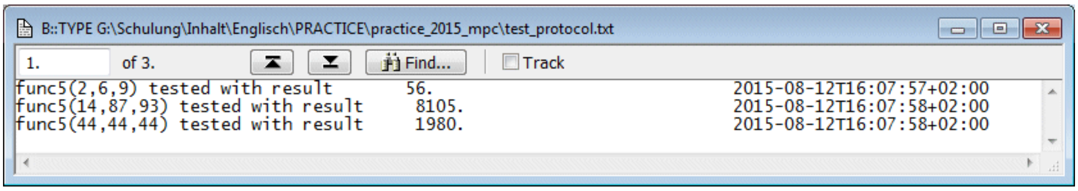
```
RePeaT <count>                Repeat <block> <count>-times.
(
  <block>
)
```

Test Functions

Task of the script: Test functions with specified parameters and generate a test protocol.

```
// Script test_function.cmm
...
PRIVATE &result

&result=FORMAT.DECIMAL(8.,Var.VALUE(func5(2,6,9)))
APPEND "test_protocol.txt" \
FORMAT.STRing("func5(2,6,9) tested with result &result",70.,' ') \
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())

&result=FORMAT.DECIMAL(8.,Var.VALUE(func5(14,87,93)))
APPEND "test_protocol.txt" \
FORMAT.STRing("func5(14,87,93) tested with result &result",70.,' ') \
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())


&result=FORMAT.DECIMAL(8.,Var.VALUE(func5(44,44,44)))
APPEND "test_protocol.txt" \
FORMAT.STRing("func5(44,44,44) tested with result &result",70.,' ') \
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
...
```

Results for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt
1.
of 3.
Find...
Track

```
func5(2,6,9) tested with result       56.                          2015-08-12T16:07:57+02:00
func5(14,87,93) tested with result     8105.                       2015-08-12T16:07:58+02:00
func5(44,44,44) tested with result     1980.                       2015-08-12T16:07:58+02:00
```

PRACTICE function

FORMAT.DECIMAL(*<width>,<value>***)**	Formats a numeric expression to a decimal number and generates an output string with a fixed length of *<width>* with leading spaces. Numeric expressions which need more characters than *<width>* for their loss-free representation aren't cut.
Var.VALUE(*<function_call>***)**	Returns the return value of called function as hex. number.

Test Function with Parameter File

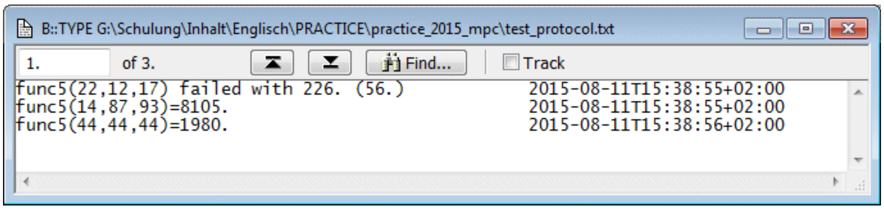
Task of script: Test functions, but provide function name, parameters and expected result by a parameter file. Generate a test protocol.

```
// Script test_func_param.cmm

LOCAL &testfunc &correct_result

OPEN #1 "func_test.txt" /READ

WHILE TRUE()
(
  READ #1 &testfunc &correct_result
  IF "&testfunc"!=""
  (
  GOSUB perform_test
  )
  ELSE
  (
   CLOSE #1
   ENDDO
  )
)
ENDDO

perform_test:
(
  IF Var.VALUE(&testfunc)==&correct_result
  (
    APPEND "test_protocol.txt"\
    FORMAT.STRing("&testfunc=&correct_result",50.,' ')\
    FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  )
  ELSE
  (
    PRIVATE &result
    &result=CONVert.HEXTOINT(Var.VALUE(&testfunc))
    APPEND "test_protocol.txt"\
    FORMAT.STRing("&testfunc failed with &result (&correct_result)",50.,' ')\
    FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  )
  RETURN
)
```

Example for a parameter file.

```
func5(22,12,17) 56.
func5(14,87,93) 8105.
func5(44,44,44) 1980.
```

Results for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_protocol.txt

1. of 3.

```
func5(22,12,17) failed with 226. (56.)            2015-08-11T15:38:55+02:00
func5(14,87,93)=8105.                             2015-08-11T15:38:55+02:00
func5(44,44,44)=1980.                             2015-08-11T15:38:56+02:00
```

PRACTICE command

GOSUB *<label>*	Call a subroutine. The start of the subroutine is identified by *<label>*. Labels must start in the first column of a line and end with a colon. No preceding white space allowed. Subroutines are usually located after the ENDDO statement.
RETURN	Return from subroutine.
LOCAL *<macro>*	Creates a local PRACTICE macro. Local PRACTICE macros are visible inside the declaring block, in all called scripts and in all called subroutines. They are erased when the declaring block ends. The declaring block is here the script itself.
WHILE *<condition>* **(** *<block>* **)**	Execute *<block>* as long as *<condition>* is true.

OPEN #*<buffer_number> <filename>* **/Read**	Open file *<filename>* for reading. The file is referenced by its **#***<buffer_number>* by the following commands.
READ #*<buffer_number>* *{<macro>}*	Read next line from file referenced by **#***<buffer_number>* into PRACTICE macros. Space serves as parameter separators.
CLOSE #<buffer_number>	Close file referenced by **#***<buffer_number>*.

TRACE32 function

CONVert.HEXTOINT(*<number>***)**	Convert *<number>* to a decimal number.

Structure of PRACTICE Programs

Program Elements

Commands

All commands of the TRACE32 development tools

Commands for program flow control and conditional commands

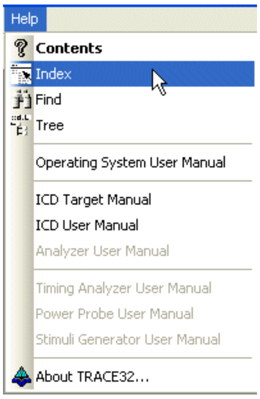
I/O commands

Functions

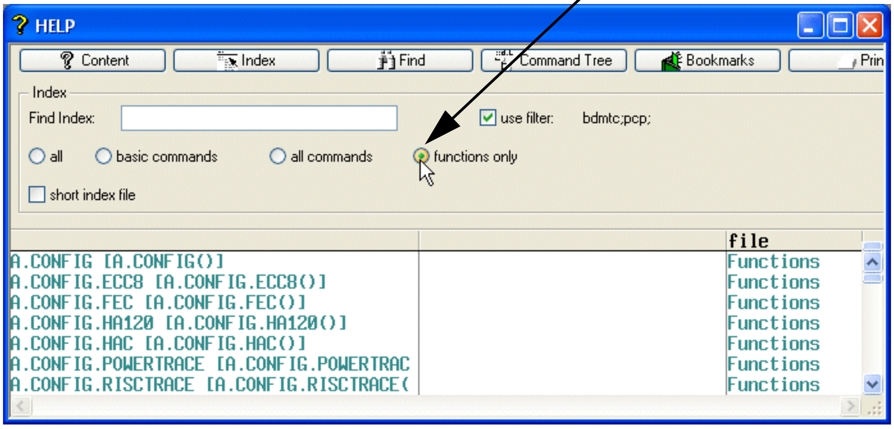
Functions are used to get information about the state of the target system or the state of the development tool.

Register(<name>)	Get the content of the specified CPU register.
V.VALUE(<hll expression>)	Get the contents of a hll expression.
STATE.RUN()	Returns true if program is running on the target, returns false if program execution is stopped.
OS.PWD()	Returns the name of the current working directory as a string.
CONV.CHAR(<value>)	Converts an integer value to an ASCII character.

A list of all available functions can be found in the on-line help:



Select **functions only**



Index

Find Index:

[✓] use filter: bdmtc;pcp;

○ all ○ basic commands ○ all commands [✓ functions only]

☐ short index file

	file
A.CONFIG [A.CONFIG()]	Functions
A.CONFIG.ECC8 [A.CONFIG.ECC8()]	Functions
A.CONFIG.FEC [A.CONFIG.FEC()]	Functions
A.CONFIG.HA120 [A.CONFIG.HA120()]	Functions
A.CONFIG.HAC [A.CONFIG.HAC()]	Functions
A.CONFIG.POWERTRACE [A.CONFIG.POWERTRAC	Functions
A.CONFIG.RISCTRACE [A.CONFIG.RISCTRACE(	Functions

CPU()	Returns the name of the selected processor as a string

```
PRINT CPU()

IF (CPU()=="TC1796")
     &int_flsh_size=0x00200000
ELSE IF (CPU()=="TC1766")
     &int_flsh_size=0x00178000
…
```

SYSTEM.UP()	Returns TRUE if the communication between the debugger and the CPU is active
STATE.RUN()	Returns TRUE if the CPU is executing the application program

```
IF !SYSTEM.UP()
     SYStem.Up

IF STATE.RUN()
     BREAK
```

VERSION.BUILD()	Returns the build number

```
 IF (VERSION.BUILD()<1146.)
     PRINT %ERROR "The version of TRACE32 too old!"
```

Data.Byte (*<address>*)	Returns byte at *<address>*
Data.Word (*<address>*)	Returns word at *<address>*
Data.Long (*<address>*)	Returns long at *<address>*

```
PRINT Data.Long(D:0xd00000008)
```

STRING.UPR(*<string>*)	Returns *<string>* converted to upper case

```
; script function_string.cmm

DIALOG.FILE *.sre
ENTRY &filename
&filename=STRING.UPR("&filename")

PRINT "&filename"

ENDDO
```

Comments

Comments start with “//”or “;”

The “;” has a special meaning with the **Var** command group.

```
Var.View flags[3];ast.count;i
```

Since the command **SETUP.Var %SPace.ON** allows spaces in variable expressions, “;” operates as a separator of variable expressions.

Labels

Labels have to be entered at the first column and they end with “:”

⚠	It is recommended to avoid unnecessary blanks. Unnecessary blanks can lead to a misinterpretation of PRACTICE commands.

```
&i = 7.                                   ; unnecessary blanks can lead to
                                          ; misinterpretations

&i=7.
```

Start a PRACTICE Program in another PRACTICE Program

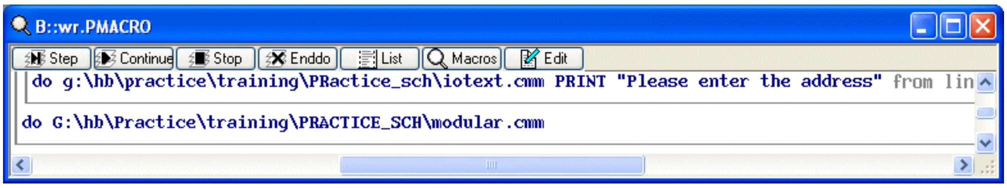
A PRACTICE program can call another PRACTICE program. This allows you to structure your PRACTICE programs.

DO *<filename>* [*<parlist>*]	Start a PRACTICE program in another PRACTICE program

```
 ; PRACTICE file modular.cmm

AREA.Create IO-AREA                         ; Create and open IO window
AREA.Select IO-AREA
WinPOS ,,,,,, IO1
AREA.view IO-AREA

DO iotext                                   ; Start PRACTICE script iotext

AREA.RESet                                  ; Close IO window
WinCLEAR IO1

ENDDO
```

The following abbreviations support the writing of movable PRACTICE scripts:

Windows	function
.\	current directory
..\	parent directory
~\	home directory of user (from $HOME)
~~\	system directory of TRACE32
~~\	temporary directory for TRACE32
~~\	directory where the current PRACTICE script is located



```
do g:\hb\practice\training\PRactice_sch\iotext.cmm PRINT "Please enter the address" from lin
do G:\hb\Practice\training\PRACTICE_SCH\modular.cmm
```

PRACTICE manages the PRACTICE program call hierarchy on a PRACTICE stack. Local and global PRACTICE variables are also maintained on this stack.

⚠	Each PRACTICE program should end with an **ENDDO** instruction.

ENDDO [*<param>*] Return from PRACTICE program

With the return from the PRACTICE program a result parameter can be passed.

```
; script test_status

ENDDO (0==0)                                          ; return TRUE as result
; ENDDO (1==0)                                        ; return FALSE as result
```

```
; script enddo_param
DO setup_CPU

DO test_status

ENTRY &result                                         ; Read result

IF &result                                            ; Evaluate result
     DIALOG.OK "Test passed"
ELSE
     (
     DIALOG.OK "Test failed"
     ENDDO
     )
ENDDO
```

More Commands

GOSUB *<label>* [*<parlist>*]	Call a PRACTICE subroutine
RETURN [*<parlist>*]	Return from a PRACTICE subroutine
GOTO *<label>*	Branch within the same PRACTICE program
JUMPTO *<label>*	Branch to a label within another PRACTICE program

ENTRY *<parlist>*

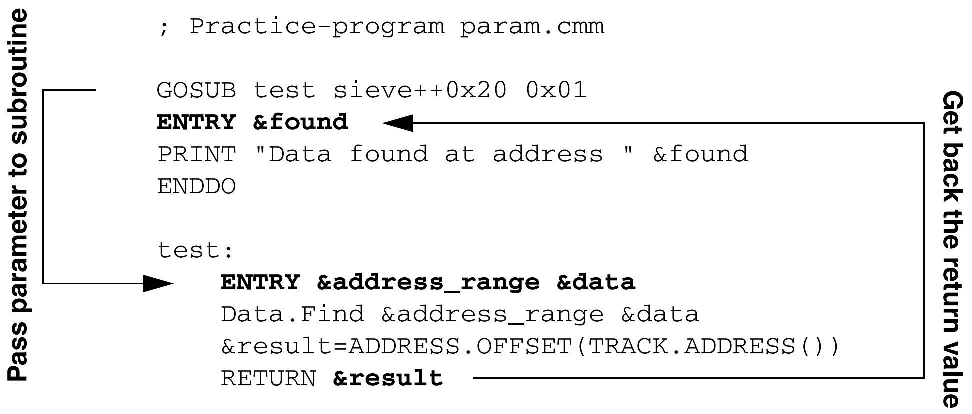
The ENTRY command can be used to

- Pass parameters to a PRACTICE program or to a subroutine
- To return a value from a subroutine

Example 1: Pass parameters to a PRACTICE program

```
; Practice program patch.cmm
; DO patch.cmm 0x1000++0xff 0x01 0x0a

ENTRY &address_range &data_old &data_new

IF "&address_range"==""
(
     PRINT "Address range parameter is missing"
     ENDDO
)

IF "&data_old"==""
(
     PRINT "Old data parameter is missing"
     ENDDO
)

IF "&data_new"==""
(
     PRINT "New data parameter is missing"
     ENDDO
)

Data.Find &address_range &data_old

IF FOUND()
(
     Data.Set TRACK.ADDRESS() &data_new
     Data.Print TRACK.ADDRESS()
     DIALOG.OK "Patch done"
)
ELSE
     DIALOG.OK "Patch failed"

ENDDO
```

Example 2: Pass parameters to a subroutine and get back the return value.



```
; Practice-program param.cmm

GOSUB test sieve++0x20 0x01
ENTRY &found
PRINT "Data found at address " &found
ENDDO

test:
    ENTRY &address_range &data
    Data.Find &address_range &data
    &result=ADDRESS.OFFSET(TRACK.ADDRESS())
    RETURN &result
```

Conditional Program Execution

For detailed information on logical operations refer to **chapter “Operators”** in **”IDE User’s Guide”** (ide_user.pdf).

IF/ELSE

IF *<condition>*
  <block>
[ELSE
 ***<block>*]**

Example:

```
; PRACTICE program patch.cmm

ENTRY &address_range &data_old &data_new

Data.Find &address_range &data_old

IF FOUND()
(
     Data.Set TRACK.ADDRESS() &data_new
     Data.Print TRACK.ADDRESS()
     DIALOG.OK “Patch done”
)
ELSE
     DIALOG.OK "Patch failed"
ENDDO
```

Block structure is similar to C.

“(“ and “)” have to be in a separate line.

IF must be followed by space.

Var.IF <hll-*condition*>
<block>
[ELSE
***<block>*]**

Example:

```
Var.IF (flags[0]==flags[5])
     PRINT "Values are equal"
ELSE
     PRINT "Values are not equal"
ENDDO
```

WHILE [<*condition*>]
<block>

Var.WHILE [<*hll-condition*>]
<block>

RePeaT [<*count*>]
<block>

RePeaT
<block>
[WHILE [<*condition*>]**]**

PRACTICE Macros

Macros are the variables of the script language PRACTICE. They works as placeholders for a sequence of characters.

Macro names in PRACTICE always start with an **&** sign, followed by a sequence of letters (a-z, A-Z) and numbers (0-9); you can also use the _ symbol in macro names. It is recommended not to use macro names which start with a number after the initial & sign.

Macro names are case sensitive, so &a is different from &A.

Create a Macro

Empty PRACTICE macros **can** be created along with their scope by using one of the following PRACTICE commands:

PRIVATE {*<macro>*}	Create PRIVATE macros. PRIVATE macros are visible in the script, subroutine or block in which they are created. And they are removed when the script, subroutine or block ends. They are visible in nested blocks, but not in called subroutines and called scripts.
LOCAL {*<macro>*}	Create LOCAL macros. LOCAL macros are visible in the script, subroutine or block in which they are created. And they are removed when the script, subroutine or block ends. They are visible in nested blocks, in called subroutines and called scripts.
GLOBAL {*<macro>*}	Create GLOBAL macros. GLOBAL macros are visible to all scripts, subroutines and blocks until they are explicitly removed by the command **PMACRO.RESet**. They can only be removed if no script is running.

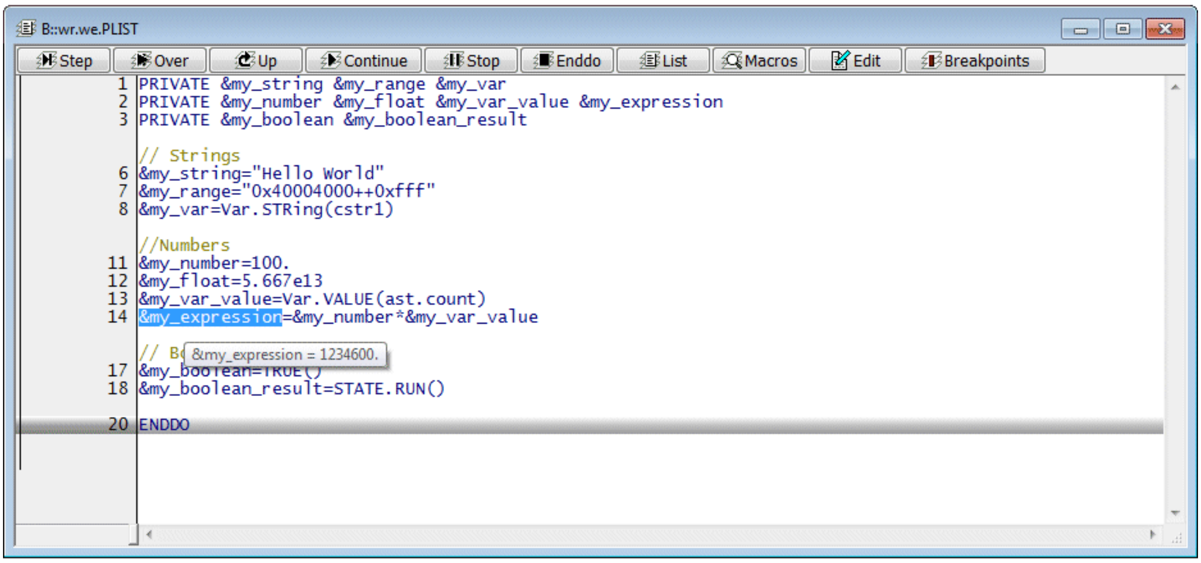
```
// Script test_func_param.cmm

//LOCAL &testfunc &correct_result

OPEN #1 "func_test.txt" /READ

WHILE TRUE()
(
  READ #1 &testfunc &correct_result
  IF "&testfunc"!=""
  (
    GOSUB perform_test
  )
  ELSE
  (
     CLOSE #1
     ENDDO
  )
)
ENDDO

perform_test:
(
  IF Var.VALUE(&testfunc)==&correct_result
  (
    APPEND "test_protocol.txt"\
    FORMAT.STRing("&testfunc=&correct_result",50.,' ')\
    FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  )
  ELSE
  (
    PRIVATE &result
    &result=CONVert.HEXTOINT(Var.VALUE(&testfunc))
    APPEND "test_protocol.txt"\
    FORMAT.STRing("&testfunc failed with &result (&correct_result)",50.,' ')\
    FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
  )
  RETURN
)
```

The PRACTICE interpreter will create a new LOCAL macro when an assignment is done, but it cannot find a macro of that name in the current scope.

The command **PMACRO.EXPLICIT** advises the PRACTICE interpreter to generate an error if a PRACTICE macro is used in an assignment, but was not created in advance. It also advises the PRACTICE interpreter to generate an error if the same macro was intentionally created a second time within its scope.

```
// Script

PRIVATE &my_string &my_range &my_var
PRIVATE &my_number &my_float &my_var_value &my_expression
PRIVATE &my_boolean &my_boolean_result

// Assign a string
&my_string="Hello World"
&my_range="0x40004000++0xfff"
&my_var=Var.STRing(cstr1)

// Assign a numbers
&my_number=100.
&my_float=5.667e13
&my_var_value=Var.VALUE(ast.count)
&my_expression=&my_number*&my_var_value

// Assign a boolean
&my_boolean=TRUE()
&my_boolean_result=STATE.RUN()

ENDDO
```

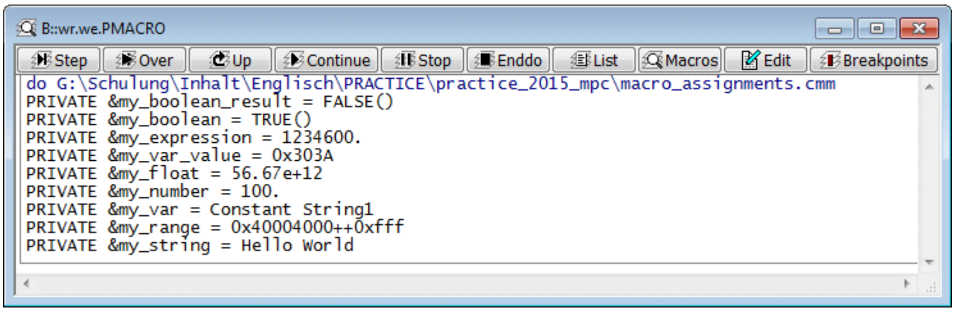
After a content is assigned to a macro, tooltips allows to inspect its current content.



```
PRIVATE &my_string &my_range &my_var
PRIVATE &my_number &my_float &my_var_value &my_expression
PRIVATE &my_boolean &my_boolean_result

// Strings
&my_string="Hello World"
&my_range="0x40004000++0xfff"
&my_var=Var.STRing(cstr1)

//Numbers
&my_number=100.
&my_float=5.667e13
&my_var_value=Var.VALUE(ast.count)
&my_expression=&my_number*&my_var_value

// B  &my_expression = 1234600.
&my_boolean=TRUE()
&my_boolean_result=STATE.RUN()

ENDDO
```

The PRACTICE stack provide an overview for all macros.

The keyword PRIVATE is used to identify PRIVATE macros.



```
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\macro_assignments.cmm
PRIVATE &my_boolean_result = FALSE()
PRIVATE &my_boolean = TRUE()
PRIVATE &my_expression = 1234600.
PRIVATE &my_var_value = 0x303A
PRIVATE &my_float = 56.67e+12
PRIVATE &my_number = 100.
PRIVATE &my_var = Constant String1
PRIVATE &my_range = 0x40004000++0xfff
PRIVATE &my_string = Hello World
```

Var.STRING(*<hll_expression>***)**	Returns a zero-terminated string, if *<hll_expression>* is a pointer to character or an array of characters. Returns a string that represents the variable contents otherwise.

To better understand the usage of macros, it is the best to look at the way the PRACTICE interpreter executes a script line.

The PRACTICE interpreter executes a script line by line. Each line is (conceptually) processed in three steps:

1. All macros are replaced by their corresponding character sequence.
2. All expressions are evaluated.
3. The resulting command is executed.

For the following examples the command **PRINT** is used.

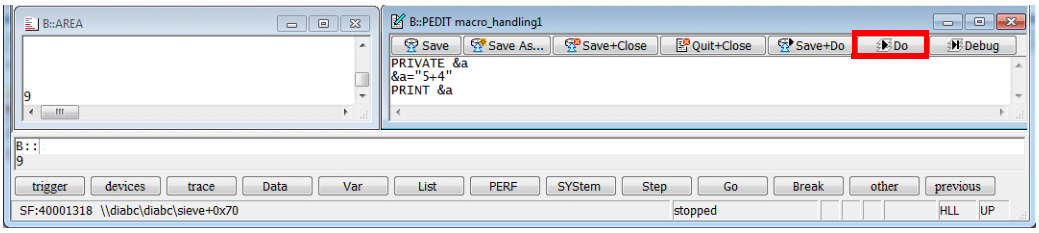
PRINT {*<data>*}	PRINT specified *<data>* to TRACE32 Message Line and to TRACE32 Message AREA.

Example 1

```
PRIVATE &a

&a="5+4"
PRINT &a
```

We look at the **`PRINT &a`** line. To execute this line, the interpreter will first:

1. Replace all macros with their corresponding character sequences. So:
   `PRINT &a -> PRINT 5+4`
2. Evaluate all expressions
   `PRINT 5+4 -> PRINT 9`
3. Execute the resulting command. So it will execute **`PRINT 9`**.



```
9
```

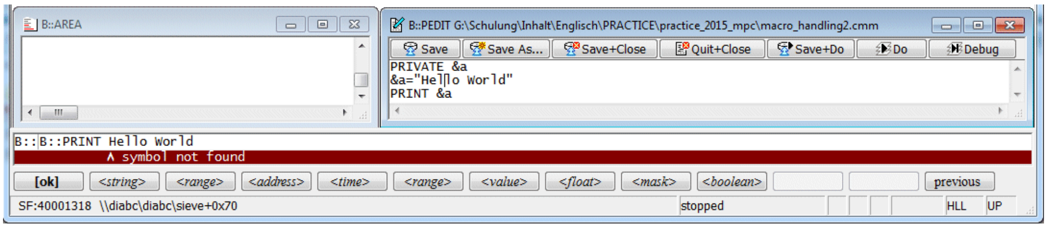
```
PRIVATE &a
&a="5+4"
PRINT &a
```

```
B::
9
```

Example 2

```
PRIVATE &a

&a="Hello World"
PRINT &a
```

This example will generate an error.



```
PRIVATE &a
&a="Hello World"
PRINT &a
```

```
B::B::PRINT Hello World
            ^ symbol not found
```

Let's look at the three steps the interpreter will take to execute the PRINT &a command:

1. Replace all macros with their corresponding character sequences
   `PRINT &a -> PRINT Hello World`

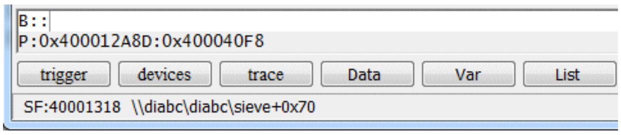
2. Evaluate expressions
   `PRINT Hello World` -> error

3. Execute command, which will not happen because of the error in the second step.

The second step fails because in PRACTICE a single word like **`Hello`** (which is not enclosed in double quotes) refers to a debug symbol, loaded for example from an ELF file.

When the PRACTICE interpreter encounters such a debug symbol, the expression evaluation will try to replace the debug symbol by the address to which the symbol refers. If there is no debug symbol called **`Hello`** (which is likely), the PRACTICE interpreter will output the error message **symbol not found**.

If by pure accident there are debug symbols called **`Hello`** and **`World`** the addresses of these symbols will be printed.



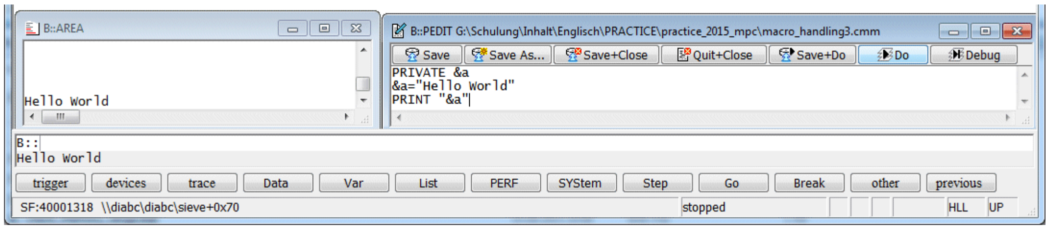
```
B::
P:0x400012A8D:0x400040F8
```

This example demonstrates how the pure macro replacement step will basically always work, since you always can replace a macro by its character sequence; but the result might not make sense.

Macros as Strings

Macros are replaced by their character sequence. If you want to explicitly use this character sequence as a string, then you should enclose the macro in double quotes, for example:

```
PRIVATE &a

&a="Hello World"
PRINT "&a"
```



```
Hello World
```

```
PRIVATE &a
&a="Hello World"
PRINT "&a"
```

```
B::
Hello World
```

To understand what happens it is again best to look at the three steps which are taken to execute the **`PRINT "&a"`** command.

1. Replace the macro by its character sequences
   `PRINT "&a" -> PRINT "Hello World"`

2. Evaluate expressions.
   Nothing to do for this example.

3. Execute command.

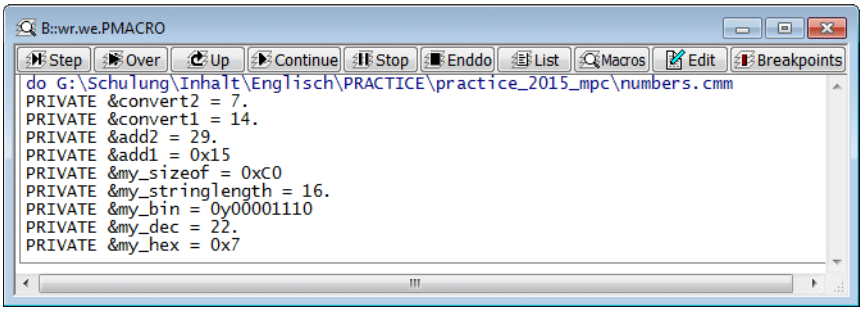
String composing example:

```
// Script string_example.cmm

PRIVATE &drive &architecture &demo_directory
&drive="C:"
&arch="MPC"

// PRINT command
PRINT "Directory " "&drive" "\T32_" "&architecture" "\demo"
PRINT "Directory "+"&drive"+"\T32_"+"&architecture"+"\demo"
PRINT "Directory &(drive)\T32_&(architecture)\demo"

// Macro assignment
&demo_directory="&drive"+"\T32_"+"&architecture"+"\demo"
DIR &demo_directory

&demo_directory="&(drive)\T32_&(architecture)\demo"
DIR &demo_directory

// Command parameter
DIR  "&(drive)\T32_&(architecture)\demo"
```

DIR *<directory>*	Display a list of files and folders for the specified directory.

```
// Script numbers.cmm

PRIVATE &my_hex &my_dec &my_bin
PRIVATE &my_stringlength &my_sizeof
PRIVATE &add1 &add2
PRIVATE &convert1 &convert2

// Hex, decimal, binary by TRACE32 syntax
&my_hex=0x7
&my_dec=22.
&my_bin=0y1110

// Hex, decimal, binary as expression result
&add1=&my_bin+&my_hex
&add2=&my_hex+&my_dec

// Hex, decimal, binary as return value
&my_stringlength=STRing.LENgth("0123456789012345")
&my_sizeof=sYmbol.SIZEOF(sieve)

// Hex, decimal, binary by CONVERT function

&convert1=CONVERT.HEXTOINT(&my_bin)
&convert2=CONVERT.HEXTOINT(&my_hex)
…
```

The PRACTICE stack shows the macro values and their radix.



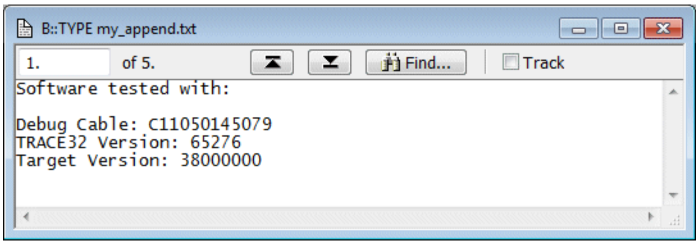
```
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\numbers.cmm
PRIVATE &convert2 = 7.
PRIVATE &convert1 = 14.
PRIVATE &add2 = 29.
PRIVATE &add1 = 0x15
PRIVATE &my_sizeof = 0xC0
PRIVATE &my_stringlength = 16.
PRIVATE &my_bin = 0y00001110
PRIVATE &my_dec = 22.
PRIVATE &my_hex = 0x7
```

STRing.LENgth(*<string>*)	Returns the length of the *<string>* as a decimal number.
sYmbol.SIZEOF(*<symbol>*)	Returns the size occupied by the specified debug *<symbol>* (e.g. function, variable, module) as a hex. number.

But if you use a PRACTICE output command, the radix information is removed.

```
// Script append_example.cmm
&PRIVATE &target_id
&target_id="D:0x40004000"

DEL "my_append.txt"

APPEND "my_append.txt" "Software tested with:"
APPEND "my_append.txt" " "
APPEND "my_append.txt" "Debug Cable: " CABLE.SERIAL()
APPEND "my_append.txt" "TRACE32 Version: " VERSION.BUILD()
APPEND "my_append.txt" "Target Version: " Data.Long(&target_id)



//...
```

Results for example in:



```
Software tested with:

Debug Cable: C11050145079
TRACE32 Version: 65276
Target Version: 38000000
```

TRACE32 command

DEL *<filename>*	Delete file specified by *<filename>*

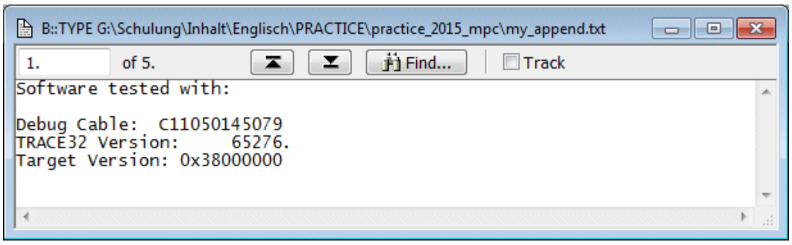
TRACE32 functions

CABLE.SERIAL()	Returns the first serial number of the plugged debug cable.
VERSION.BUILD()	Returns build number of TRACE32 software as a decimal number.

Since it might be confusing for the reader not to know if a number is decimal or hex, you can proceed as follows:

```
// Script append_example_format.cmm

&PRIVATE &target_id
&target_id="D:0x40004000"
DEL "my_append.txt"

APPEND "my_append.txt" "Software tested with:"
APPEND "my_append.txt" " "
APPEND "my_append.txt" "Debug Cable:  " FORMAT.STRING(CABLE.SERIAL(),15.,' ')
APPEND "my_append.txt" "TRACE32 Version:   " FORMAT.DECIMAL(8.,VERSION.BUILD())+"."
APPEND "my_append.txt" "Target Version: 0x"+FORMAT.HEX(8.,Data.Long(&target_id))
```

May result for example in:


B::TYPE G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\my_append.txt
1.
of 5.
Find...
Track

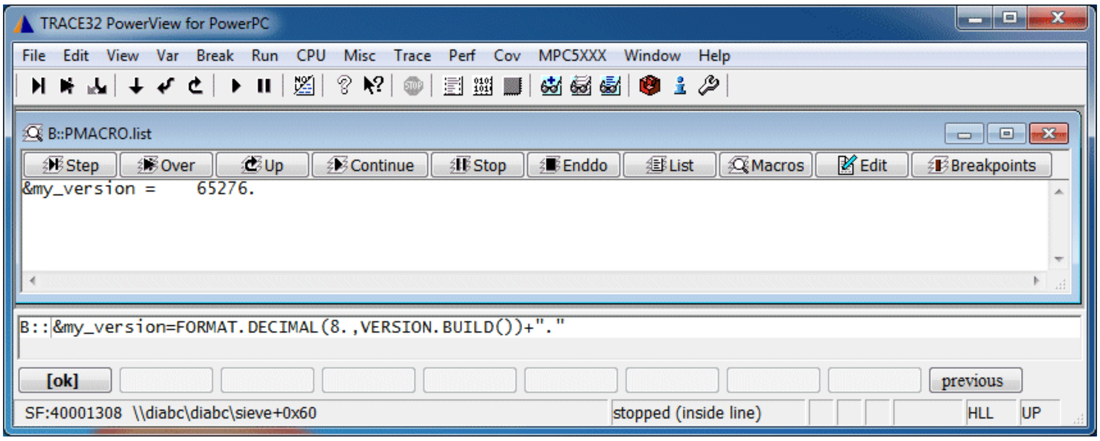
```
Software tested with:

Debug Cable:  C11050145079
TRACE32 Version:      65276.
Target Version: 0x38000000
```

TRACE32 function

FORMAT.HEX(*<width>*,*<number>***)**	Formats a numeric expression to a hexadecimal number and generates an output string with a fixed length of *<width>* with leading zeros.

Note for Testing

PRACTICE macros are not available in the command line. They are only available when running a script. But you can proceed as follow to test a macro assignment:


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX Window Help
B::PMACRO.list
Step Over Up Continue Stop Enddo List Macros Edit Breakpoints
[ok]
previous
HLL UP

```
&my_version =     65276.
```

```
B::&my_version=FORMAT.DECIMAL(8.,VERSION.BUILD())+"."
```

```
SF:40001308  \\diabc\diabc\sieve+0x60    stopped (inside line)
```

More Complex Data Structures

For all complex data structures TRACE32-internal variables can be used. The following two commands can be used to declare a TRACE32-internal variable.

Var.NEWGLOBAL *<type>* *<name>*	Create a global TRACE32-internal variable
Var.NEWLOCAL *<type>* *<name>*	Create a local TRACE32-internal variable

TRACE32-internal variables require that a program is loaded via the **Data.LOAD** command. All data types provided by this program can then be used (**sYmbol.List.Type**).

- TRACE32-internal variables have the same scope as PRACTICE macros (e.g. they are on the PRACTICE stack).
- TRACE32-internal variables are displayed and modified via the **Var** command group.

```
; script newlocal.cmm

LOCAL &my_symbol
ENTRY  &my_symbol

Var.NEWLOCAL char[5][40] \typeresult

Var.Assign \typeresult[0]="Symbol does not exist"
Var.Assign \typeresult[1]="Symbol is label"
Var.Assign \typeresult[2]="Symbol is function"
Var.Assign \typeresult[3]="Symbol is variable"
Var.Assign \typeresult[4]="Undefined"

&n=sYmbol.TYPE(&my_symbol)
Var.PRINT %String  \typeresult[&n]
ENDDO
```

Var.Assign **%**<*format*> <*variable*>	Modify variable, no log is generated in the message line and the **AREA** window.

sYmbol.TYPE(<*symbol*>**)**	Returns the basic type of the symbol as a numerical value. 0 = symbol does not exist 1 = plain label without type information 2 = HLL function 3 = HLL variable other values may be defined in the future

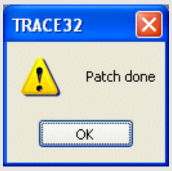
Output Command

PRINT *<format> <parlist>*

DIALOG.OK *<message>*

```
PRINT "FLASH programmed successfully"

PRINT %ERROR "FLASH programming failed"
```

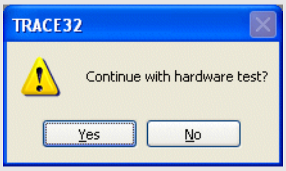
```
DIALOG.OK "Patch done"
```



Patch done

ENTER *<parlist>*	Window based input
INKEY [*<par>*]	Input command (character)
DIALOG.YESNO *<message>*	Create a standard dialog
DIALOG.File *<message>*	Read a filename via a dialog

```
INKEY                                          ; Wait for any key

INKEY &key                                     ; Wait for any key, key
                                               ; code is entered to &key

; PRACTICE program dialog.cmm
DIALOG.YESNO "Continue with hardware test?"
```



Continue with hardware test?

```
ENTRY &result

IF &result
(
     PRINT "Test started"
     DO test2
)
ELSE
     PRINT "Test aborted"
ENDDO
```

```
DIALOG.File *sre
ENTRY &filename

Data.LOAD.S3record &filename

ENDDO
```

An I/O window is needed for PRACTICE inputs and outputs. To realize this an AREA window is used.

Open and assign an AREA window

1. Create an AREA-Window

```
AREA.Create [<area> ]
```

2. Select an AREA window for PRACTICE I/O.

```
AREA.Select [<area>]
```

3. Select the screen position of the AREA window. This command is used here, because it allows you to assign a name to an AREA window. This is useful, if you want to delete this window after the I/O procedure.

```
WinPOS [<pos>] [<size>] [<scale>] [<name>] [<state>] [<header>]
```

4. Display AREA-Window

```
AREA.view [<area>]
```

Remove AREA Window

1. Resets the AREA window settings to the default settings: the message area (AREA A000) is used for error and system messages. No other AREA window is active.

```
AREA.RESet
```

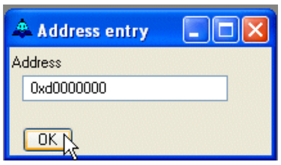
2. Delete a specific windows.

```
WinCLEAR [<pagename>|<windowname>|TOP]
```

```
; PRACTICE file iowindow.cmm

AREA.Create IO-AREA
AREA.Select IO-AREA
WinPOS ,,,,,, IO1
AREA.view IO-AREA

PRINT "Please enter the address"
PRINT "Address="
ENTER &a
PRINT " "
PRINT "Entered address=" &a

WAIT 2.s

AREA.RESet
WinCLEAR IO1

ENDDO
```

It is also possible to use the DIALOG programming feature to realize an I/O.

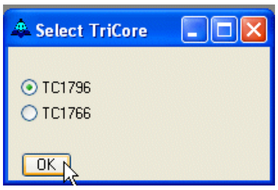
```
; PRACTICE file dialog_prog.cmm

 DIALOG
(
     HEADER "Address entry"
     POS 0. 0. 25.
     TEXT "Address"
     POS 1. 1. 10.
ADD: DEFEDIT " " " "
     POS 1. 3. 5.
     DEFBUTTON "OK" "GOTO okclose"
)
STOP

okclose:
     &address=DIALOG.STRING(ADD)
     PRINT "Entered address=" &address

DIALOG.END
ENDDO
```



STOP	Stop the execution of a PRACTICE program

In the example above the execution of the PRACTICE program is continued with the **`GOTO okclose`** command, that is assigned to the `DEFBUTTON`.

```
DIALOG
(
     HEADER "Select TriCore"                      ; Dialog header
     POS  0. 0. 25. 1.                            ; Increase dialog width
     TEXT ""                                      ; by empty text
     POS 1. 1. 10.
TC.1796:  CHOOSEBOX "TC1796" ""                   ; Define 2 choose boxes
TC.1766:  CHOOSEBOX "TC1766" ""                   ; They are exclusive
     POS 1. 4. 5.                                 ; alternatives
     DEFBUTTON "OK" "continue"
)

     DIALOG.SET TC.1796                           ; Define default setting
                                                  ; for choosebox

          STOP
     IF DIALOG.BOOLEAN(TC.1796)                   ; Evaluate result
          CPU="TC1796"

     IF DIALOG.BOOLEAN(TC.1766)
          CPU="TC1766"

DIALOG.END

PRINT "Selected CPU= " "&CPU"
ENDDO
```



For more information on the **DIALOG** programming commands refer to ***Operation System Reference***.

File Commands

File commands are used to store data into a file or to read test data from a file

- Open file

OPEN #<*buffer*> <*filename*> **/ Read** | **Write** | **Create**

- Close file

CLOSE #<*buffer*>

- Read data from an open file

READ #<*buffer*> [**%LINE**] <*parlist*>

- Write data to an open file

WRITE #<*buffer*> <*parlist*>

Example 1:

```
; PRACTICE file write.cmm

OPEN #1 abc.txt /Create

&i=0

WHILE &i<10.
(
     WRITE #1 "flags[" &i "]=" V.VALUE("flags[&i]")
     &i=&i+1
)

CLOSE #1

ENDDO
```

Example 2: Reads data from a file. Single data are separated by blank or carriage return within the file.

```
; PRACTICE file read.cmm

OPEN #2 read.txt /Read

READ #2 &e1 &e2 &e3 &e4

PRINT "&e1" " " "&e2" " " "&e3" " " "&e4"

CLOSE #2

ENDDO
```

Example 3: Read data from a file. Read always a complete line.

```
; PRACTICE file read_line.cmm

OPEN #2 read.txt /Read

READ #2 %LINE &e1

PRINT "&e1"

CLOSE #2

ENDDO
```

Event Control via PRACTICE

ON ERROR *<command>*	Perform commands on PRACTICE runtime error
ON SYSUP *<command>*	Perform commands when the communication between debugger and CPU is established
ON POWERUP *<command>*	Perform commands on target power on

```
…
ON ERROR GOTO
     (
          DIALOG.OK "Abortion by error!"
          ENDDO (0!=0)
     )
…
```

```
ON POWERUP GOTO startup

IF !(STATE.POWER())
     STOP

startup:
     SYStem.CPU TC1796
     WAIT 0.5s
     SYStem.UP

ENDDO
```

Index (local)