LAUTERBACH
DEVELOPMENT TOOLS

Training Script Language PRACTICE

MANUAL

Training Script Language PRACTICE

Training Script Language PRACTICE

Version 06-Jun-2024

History

31-Jan-2023 Solaris was removed as supported host OS.

02-Nov-2022 In the chapter ‘Related Documents‘ has been added.

E-Learning

Videos about the script language PRACTICE can be found here:
support.lauterbach.com/kb/articles/practice-tutorial

Ready-to-Run Scripts

Ready-to-run PRACTICE scripts provided by the Lauterbach experts are published and updated daily here:
https://www.lauterbach.com/scripts.html

Related Documents

- **“PowerView User’s Guide”** (ide_user.pdf): In the chapters **Operands** and **Operators** you will find everything that you need to know about operands and operators.

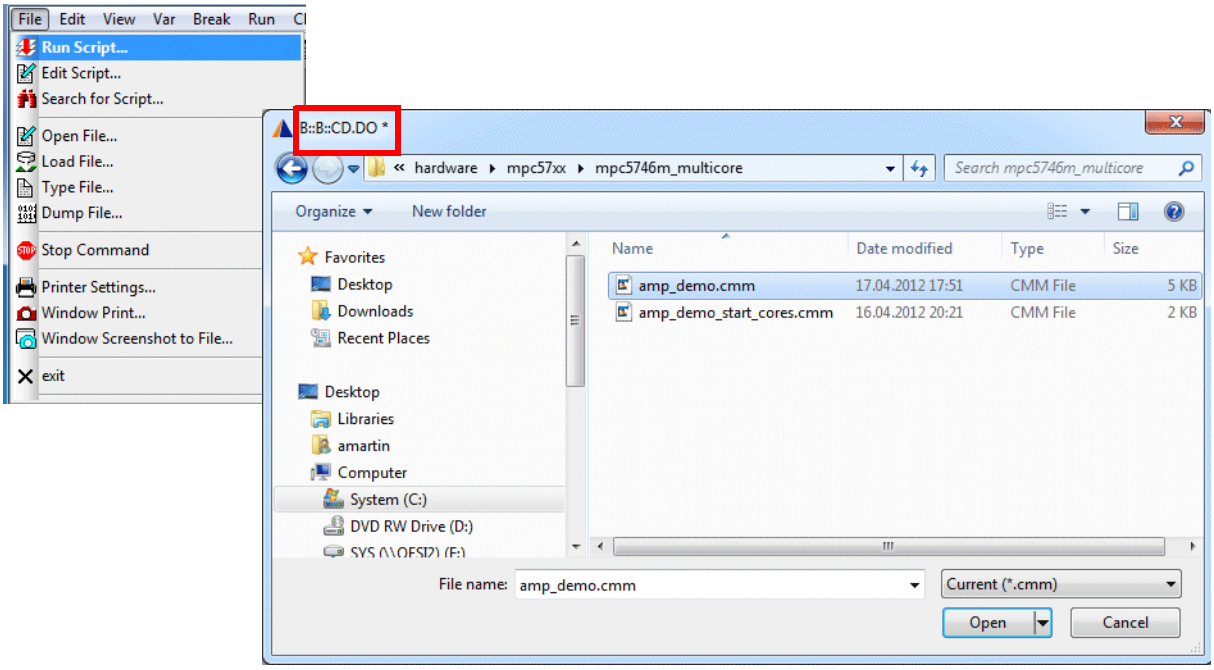
Area of Use

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




```
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
```

It is possible to execute a PRACTICE script on startup of TRACE32. For details refer to **"Automatic Start-up Scripts"** in PRACTICE Script Language User's Guide, page 14 (practice_user.pdf).

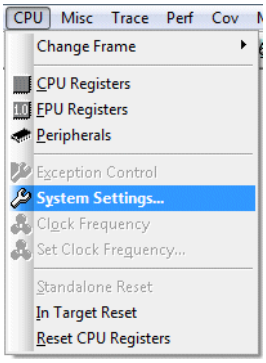
Convert TRACE32 Settings to a Script

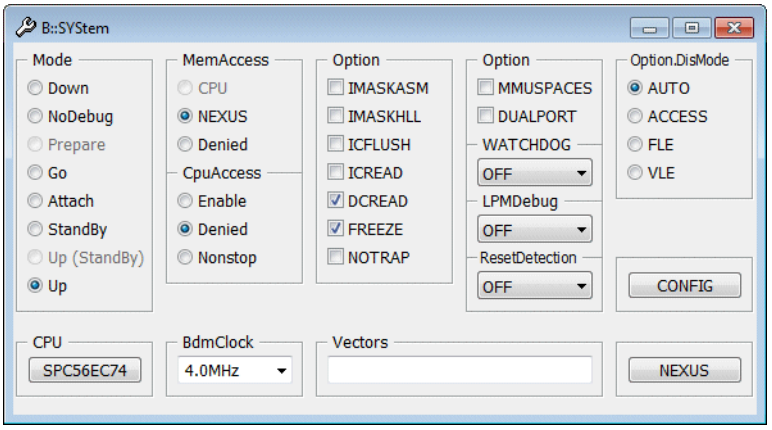
The commands **STOre** and **ClipSTOre** generate scripts that allow to reactivate the specified TRACE32 *<setting>* at any time.

<setting> is in most cases the setup of a command group.

<setting>	
SYStem	Setting for command group SYStem.
Break	Setting for command group Break.
Win	TRACE32 window configuration.

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

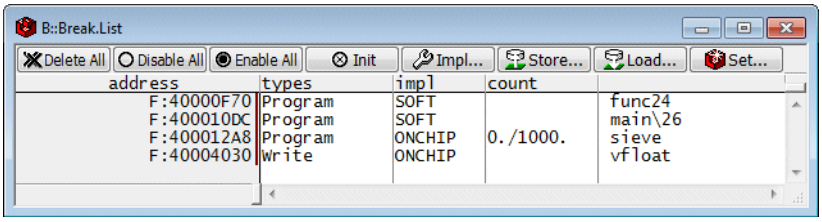





```
ClipSTOre SYStem
```

```
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
```

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

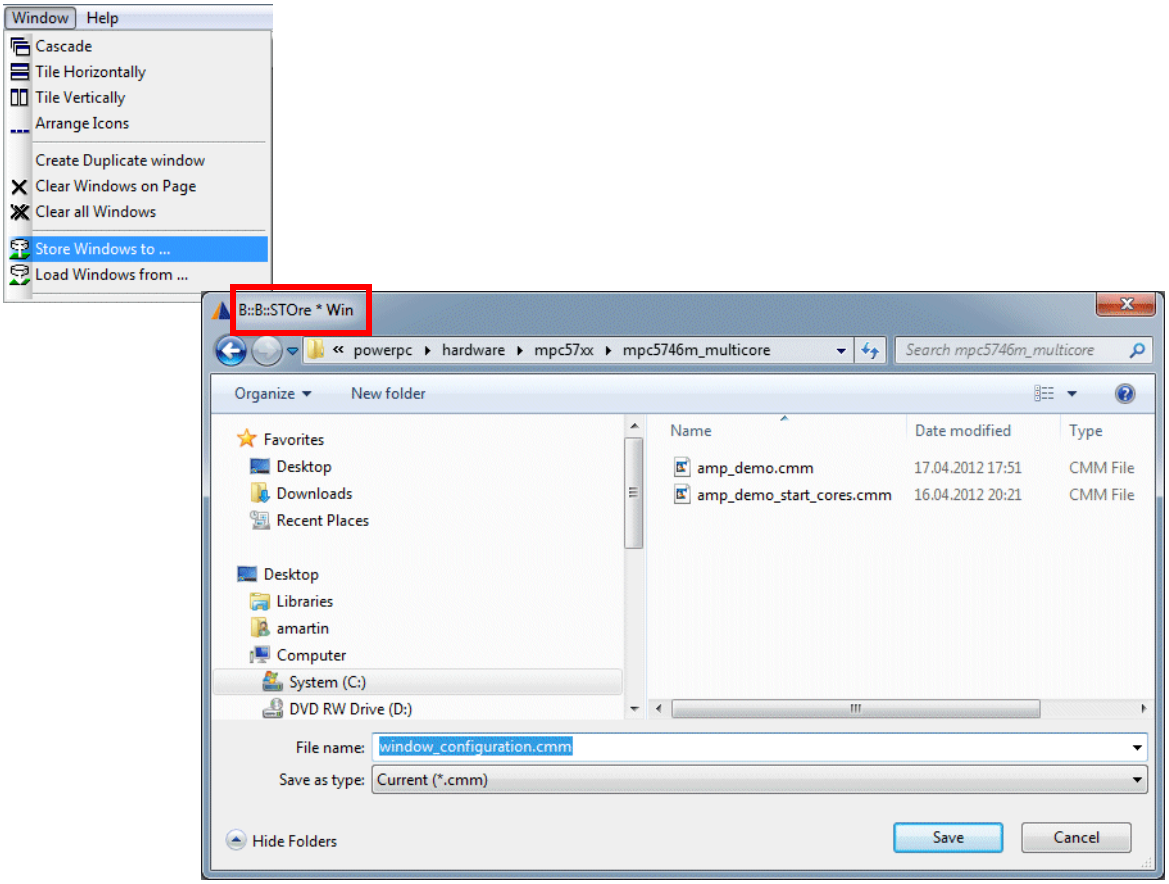



```
ClipSTOre Break
```

```
Break.RESet
Break.Set     func24 /Program
Break.Set     main\26 /Program
Break.Set     sieve /Program /Onchip /COUNT 1000.
Var.Break.Set mstatic1; /Write
```

The breakpoints are saved at a symbolic level by default.

Create a script to reactivate the current window configuration




```
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
```

.

Each PRACTICE script should end with an **ENDDO** instruction.

ENDDO Return from PRACTICE script.

Command LOG

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

Commands to control the LOG command:

LOG.OPEN *<log_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.

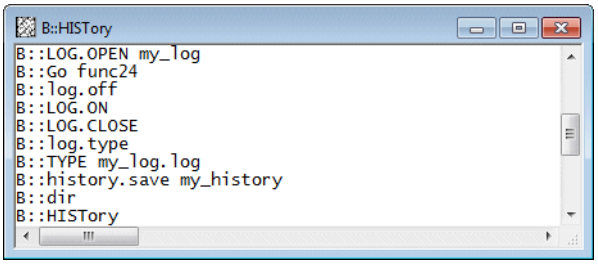
```
LOG.OPEN my_log.log    // Creates and opens the .log file

LOG.type               // 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
```

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



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

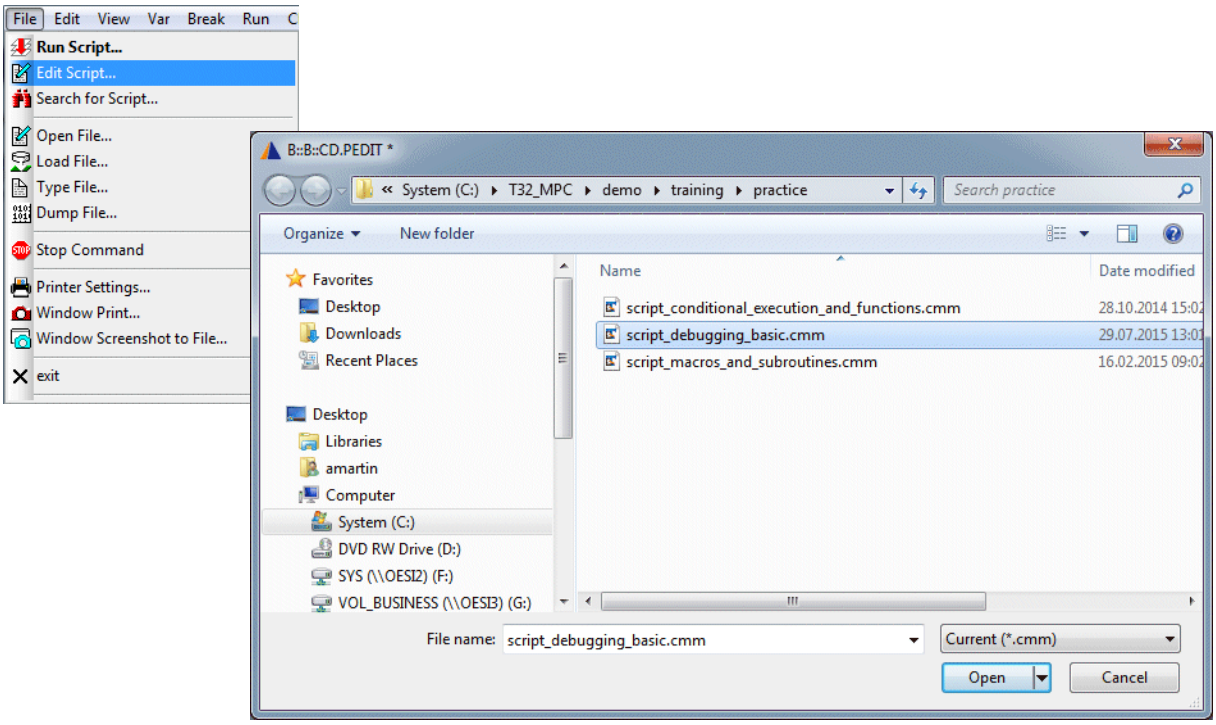
By default the script autostart.cmm contains the line:

```
AutoSTOre , HISTory
```

which automatically saves the command history in your temporary directory when you exit TRACE32 and recalls the command history when you start TRACE32 again.

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

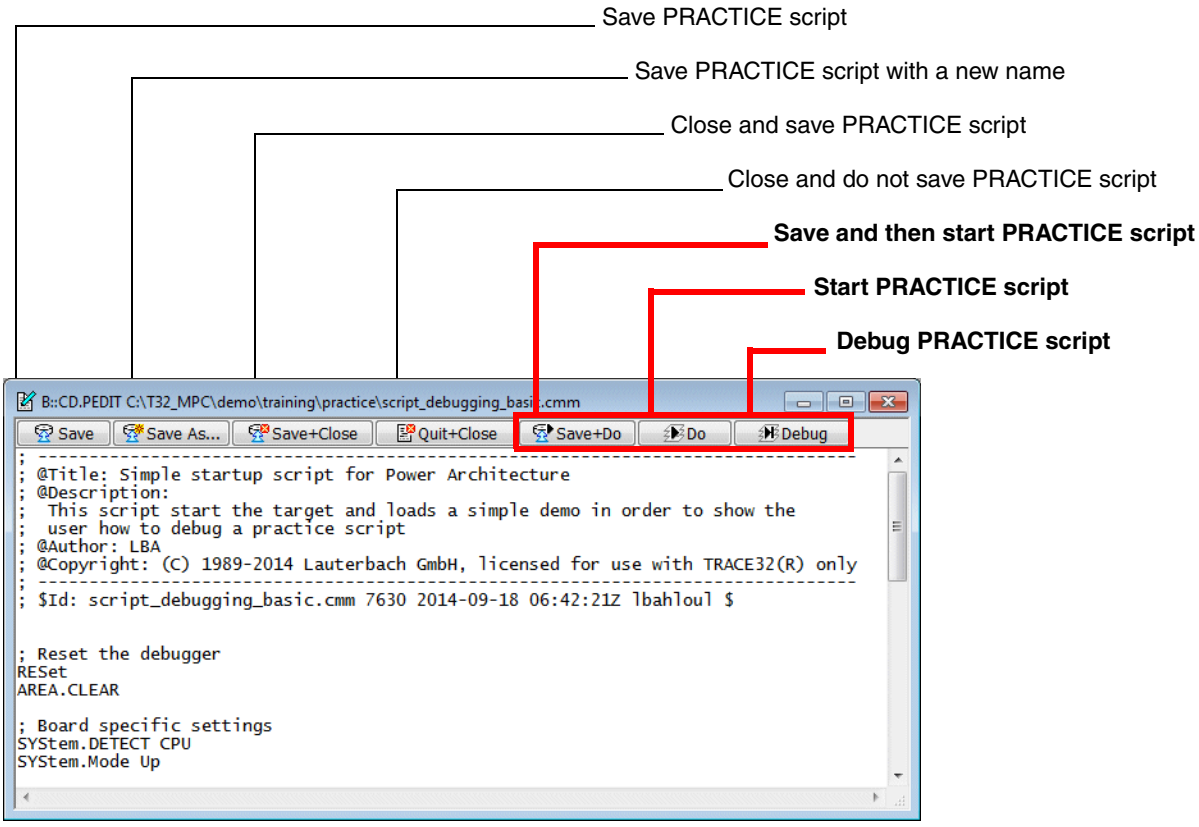
Script Editor PEDIT




```
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.

In addition to a standard editor, PEDIT provides the ability to start or to debug a script.



Syntax Highlighting

In TRACE32, the PRACTICE script editor **PEDIT** provides syntax highlighting, configurable auto-indentation, multiple undo and redo.

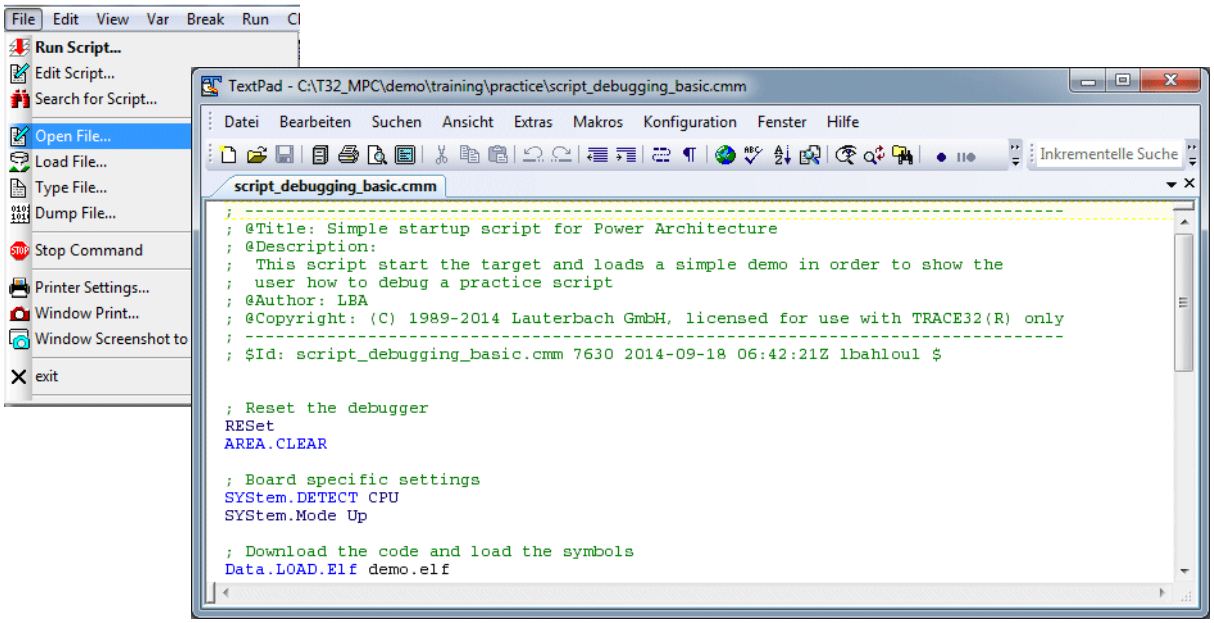
In addition to, or as an alternative to the built-in PRACTICE script editor **PEDIT**, you can work with an external editor. To configure syntax highlighting for PRACTICE scripts in an external editor, take these steps:

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

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



Debugging of PRACTICE Script

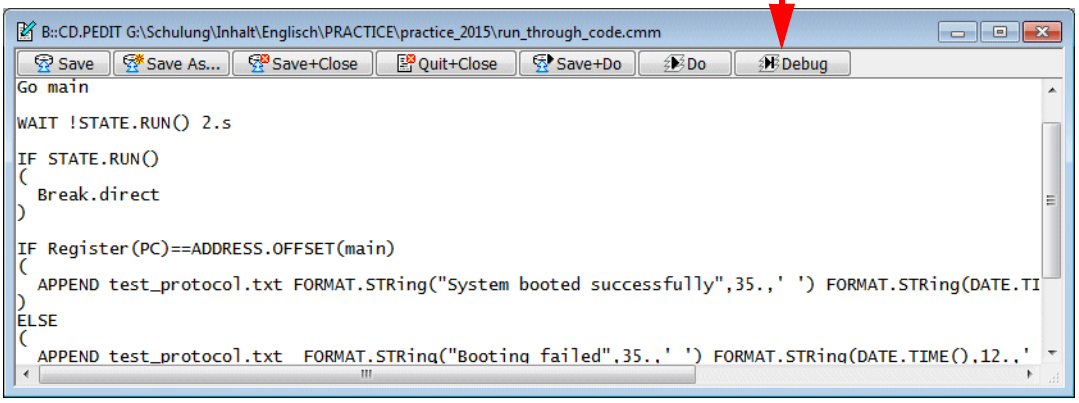
TRACE32 supports the debugging of PRACTICE scripts.

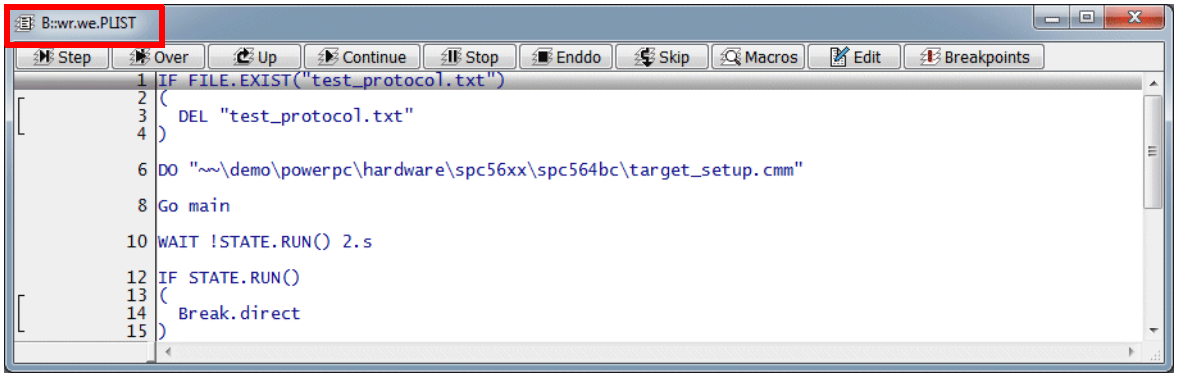
A short video that provides an introduction into PRACTICE debugging is available on:

support.lauterbach.com/kb/articles/practice-tutorial

Debug Environment

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



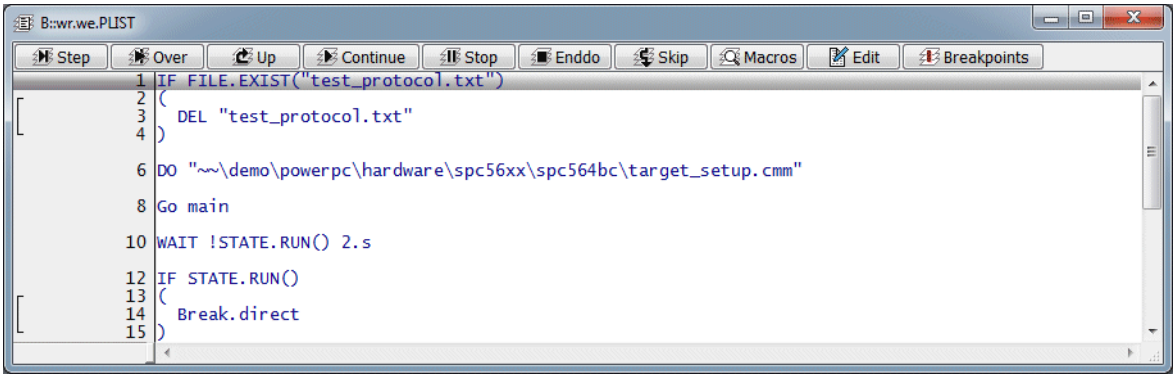


Explanation of the window header:

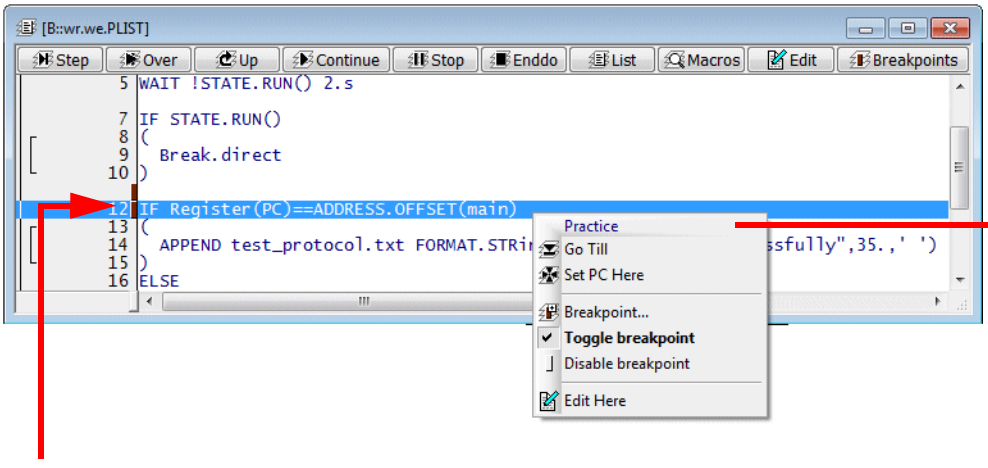
WinResist.WinExt.PLIST	Display the currently loaded PRACTICE script. **WinResist**: PEDIT window is not deleted by command **WinCLEAR.** **WinExt**: Detach PEDIT window from the TRACE32 main window - even if TRACE32 is operating in MDI window mode.

Command line commands:

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>*	



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**).
Skip	Skips the current command or block (command **PSKIP**)
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.



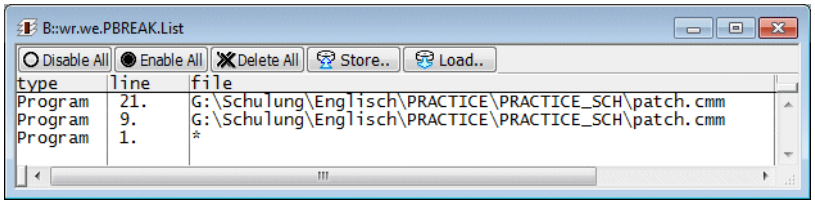
PRACTICE breakpoint
in this line

Right-click for PRACTICE
debug-pull-down

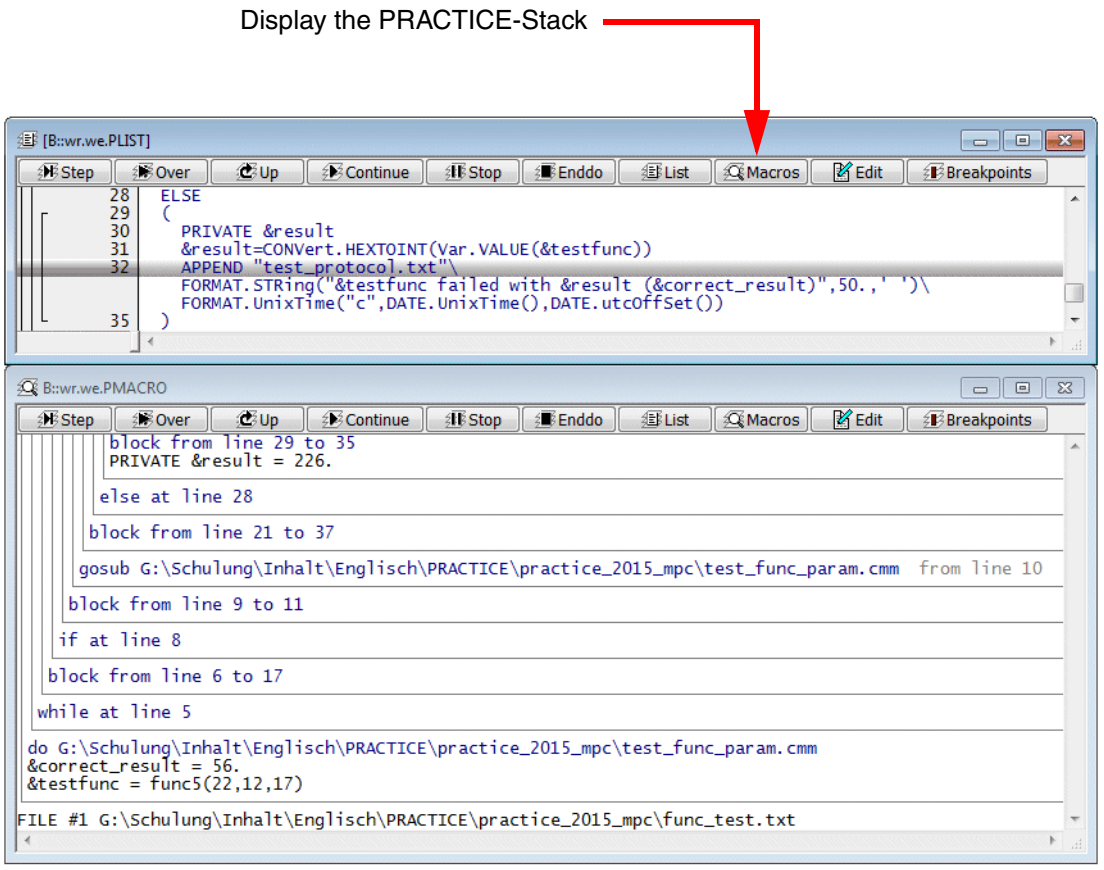
PRACTICE debug-pull-down	
Goto Till	Run PRACTICE script until the selected line (command **CONTinue** *<line_number>*).
Set PC Here	Set PRACTICE PC to the selected line.
Breakpoint …	Open **PBREAK.Set** dialog to configure PRACTICE breakpoint.
Toggle breakpoint	Toggle PRACTICE breakpoint.
Disable breakpoint	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 (*)



Display the PRACTICE Stack



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

PRACTICE Language

So far, we have seen that simple scripts can be created that can restore various settings to the current debug session. PRACTICE provides a lot more capability than this; it is a fully featured language with its own syntax and advanced features. This section will cover the language syntax, program elements, features and functions. Examples of PRACTICE scripts can be found in `~~/demo/practice` or at

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

Program Elements

Comments

Comments start with **//** or **;** and end with the next line break.

Since the TRACE32 hll expression grammar allows **;** to end an hll expression, **;** has a special meaning for the **Var** command group. Here a few examples.

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

Var.Break.Set flags[3]; /Write /VarCONDition (flags[12]==0)
```

So to be safe it is recommended to use **//** to start a comment.

Commands

There are three types of commands that can be used in PRACTICE scripts:

- All TRACE32 commands which are also used interactively in the command line
- Commands for flow control and conditional execution of PRACTICE scripts.
- I/O commands

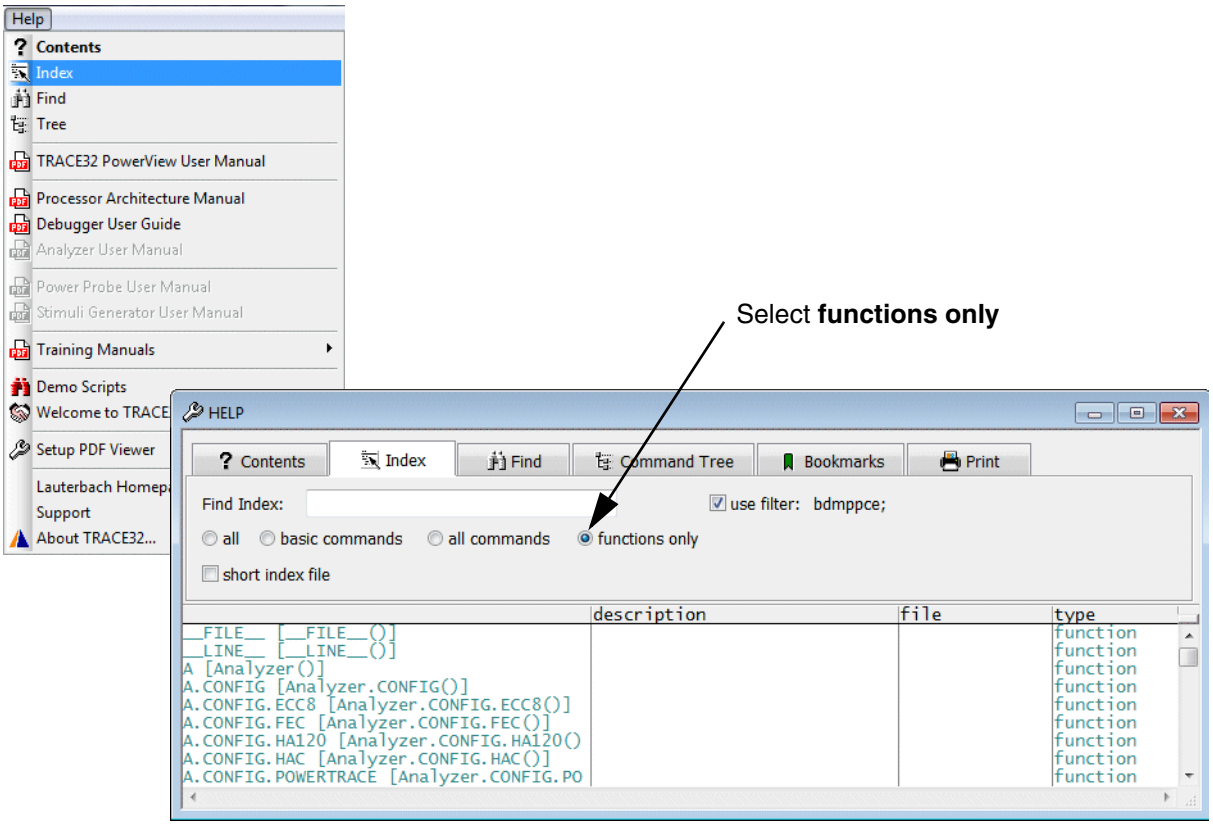
Functions

Functions are used to retrieve information about the state of the target system or the state of the development tool(s).

Register(<register_name>)	Get the content of the specified CPU register.
Var.VALUE(<hll_expression>)	Get the contents of an HLL expression.

STATE.RUN()	Returns true if program is running on the target, returns false if program execution is stopped.
OS.PresentWorkingDirectory()	Returns the name of the current working directory as a string.
CONVert.CHAR(<value>)	Converts an integer value to an ASCII character.

A list of all available functions can be found in the online help:



Labels

Labels have to be entered in the first column and they end with ":". For more information on labels please refer to the section on **subroutines**.

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

Conditional Program Execution

Command List

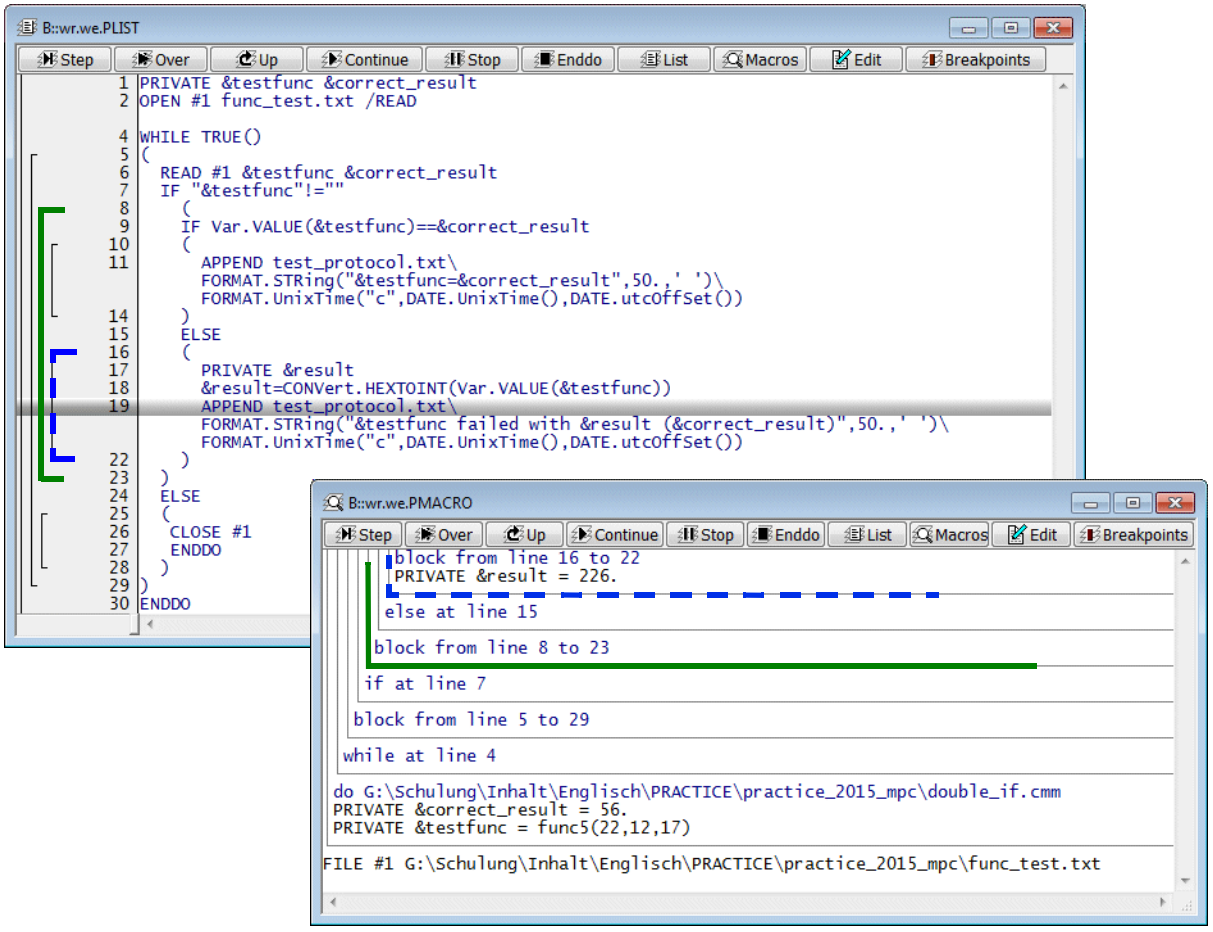
IF *<condition>* **(** *<if_block>* **)** **ELSE** **(** *<else_block>* **)**	Execute *<if_block>* if *<condition>* is true. Execute *<else_block>* otherwise. *<condition>* has to be specified in **TRACE32 syntax**.

```
// Script double_if.cmm

PRIVATE &testfunc &correct_result
OPEN #1 "func_test.txt" /READ

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

If script lines are too long they can be split by adding a space and the '\' character at the end.

To better understand the scope of PRACTICE macros given later in this training, it is important to know how TRACE32 maintains information on the PRACTICE stack.

TRACE32 adds a new **block-frame** to the PRACTICE stack, whenever instructions are blocked by round brackets.



PLIST	List PRACTICE script that is currently being executed.
PMACRO.list	List PRACTICE stack.

Var.IF *<hll_condition>*
(
<if_block>
)
ELSE
(
<if_block>
)

Execute *<if_block>* if *<hll_condition>* is true. Execute *<else_block>* otherwise.

<hll_condition> has to be written in the syntax of the programming language used.

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

WHILE *<condition>*
(
<block>
)

Execute *<block>* while *<condition>* is true.

<condition> has to be specified in **TRACE32 syntax**.

Var.WHILE *<hll_condition>*
(
<block>
)

Execute *<block>* while *<hll_condition>* is true.

<hll_condition> has to be written in the syntax of the programming language used.

RePeaT *<count>* *<command>*	Repeat *<command>* *<count>*-times.
RePeaT *<count>* **(** *<block>* **)**	Repeat *<block>* *<count>*-times.
RePeaT **(** *<block>* **)** **WHILE** *<condition>*	Repeat the *<block>* whilst the *<condition>* evaluates to a boolean TRUE.

```
;Example 1
;Print the character X 100 times
AREA.view
RePeaT 100. PRINT "X"
```

```
;Example 2
Var.Break.Set flags /Write              //Set a Write breakpoint to array
                                        //flags

;Repeat the following 10 times
;Start the program and wait until the target halts at the breakpoint.
;Then export the contents of array flags to file flags_export.csv in CSV
;format.
RePeaT 10.
(
    Go
    WAIT !STATE.RUN()
    Var.EXPORT "flags_export.csv" flags /Append
)
```

```
;Example 3
;Read a line from my_strings.txt
;Write not-empty lines to file my_strings_noempty.txt
PRIVATE &CurrentLine &RightLine
OPEN #1 my_strings.txt /Read
OPEN #2 my_strings_noempty.txt /Create
AREA.view
RePeaT
(
    READ #1 %LINE &CurrentLine
    IF (!FILE.EOFLASTREAD()&&("&CurrentLine"!=””))
        WRITE #2 "&CurrentLine"
)
WHILE !FILE.EOFLASTREAD()
CLOSE #1
CLOSE #2
```

Command List

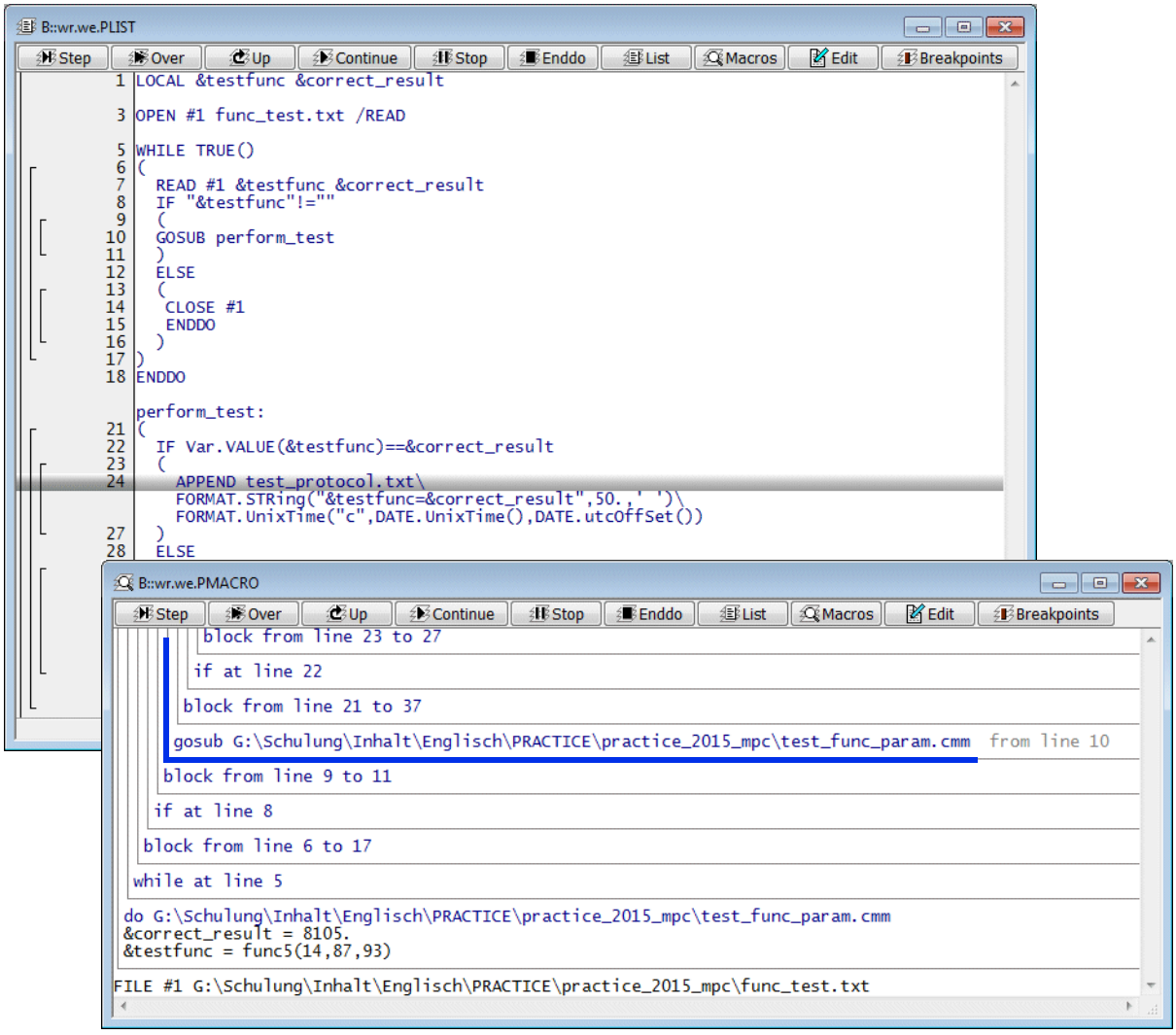
GOSUB *<label>* [*<parameter_list>*]	Call subroutine specified by *<label> with an optional set of paramters..*
...	
ENDDO	
<label>	
(	
<block>	
RETURN	Execute *<block>* and RETURN to caller.
)	

NOTE:	Labels must start in the first column of a line and end with a colon. No preceding white space allowed.

Example

```
// 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
)
```

TRACE32 adds a new **gosub-frame** to the PRACTICE stack whenever a subroutine is called.



Command List

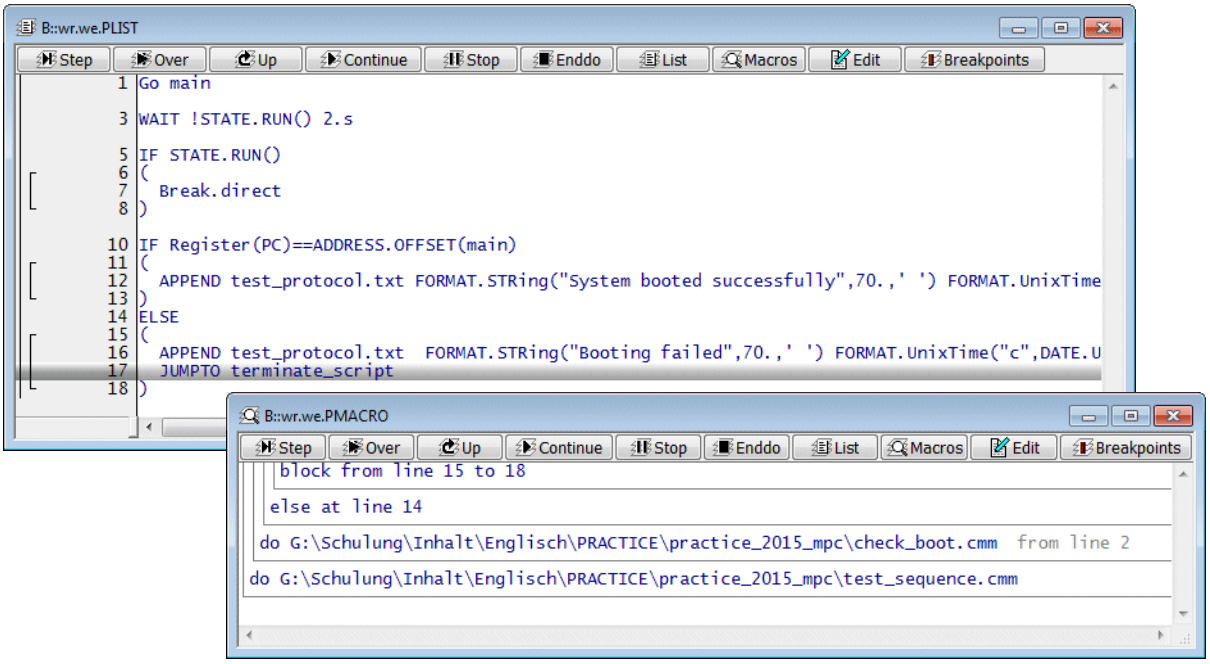
GOTO *<label>*	Continue PRACTICE script at *<label>*. *<label>* must be part of the currently executing script.
JUMPTO *<label>*	Continue PRACTICE script at *<label>*. *<label>* must be part of a script that is currently located on the PRACTICE stack. *<label>* must not be located in a block. The PRACTICE stack is cleaned up accordingly.

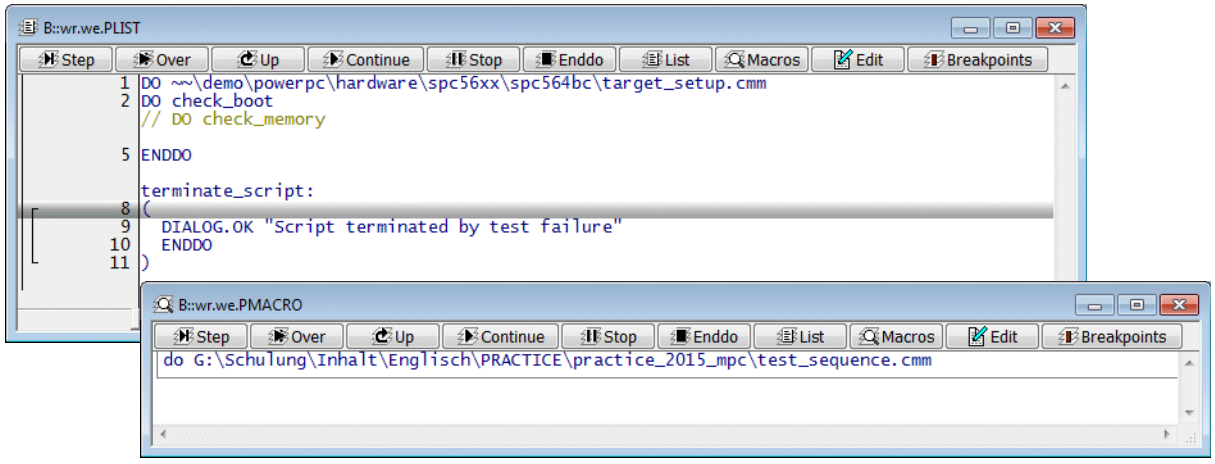
Example

```
// Script test_sequence.cmm

DO target_setup.cmm
DO check_boot.cmm


ENDDO

terminate_script:
(
  DIALOG.OK "Script terminated by test failure"
  ENDDO
)
```

```
// Script check_boot.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())
  JUMPTO terminate_script
)
```

DIALOG.OK *<text>*	Create a dialog that provides a *<text>*-message to the user. The script execution is stopped until the user pushes the OK button in the dialog.


B::wr.we.PLIST
Step
Over
Up
Continue
Stop
Enddo
List
Macros
Edit
Breakpoints
1 Go main
3 WAIT !STATE.RUN() 2.s
5 IF STATE.RUN()
6 (
7 Break.direct
8)
10 IF Register(PC)==ADDRESS.OFFSET(main)
11 (
12 APPEND test_protocol.txt FORMAT.STRing("System booted successfully",70.,' ') FORMAT.UnixTime
13)
14 ELSE
15 (
16 APPEND test_protocol.txt FORMAT.STRing("Booting failed",70.,' ') FORMAT.UnixTime("c",DATE.U
17 JUMPTO terminate_script
18)
B::wr.we.PMACRO
block from line 15 to 18
else at line 14
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\check_boot.cmm from line 2
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_sequence.cmm

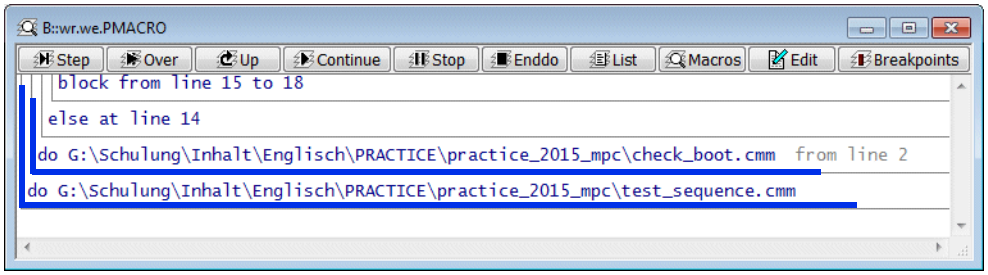

B::wr.we.PLIST
Step
Over
Up
Continue
Stop
Enddo
List
Macros
Edit
Breakpoints
1 DO ~~\demo\powerpc\hardware\spc56xx\spc564bc\target_setup.cmm
2 DO check_boot
// DO check_memory
5 ENDDO
terminate_script:
8 (
9 DIALOG.OK "Script terminated by test failure"
10 ENDDO
11)
B::wr.we.PMACRO
do G:\Schulung\Inhalt\Englisch\PRACTICE\practice_2015_mpc\test_sequence.cmm

Command List

DO *<script>* [*<parameter_list>*]	A script can be started from the command line or called within a script. Optional parameters can be passed to the script.

Example

```
// Script test_sequence.cmm

DO target_setup.cmm
DO check_boot.cmm


ENDDO

terminate_script:
(
  DIALOG.OK "Script terminated by test failure"
  ENDDO
)
```



TRACE32 adds a new **do-frame** to the PRACTICE stack whenever a script is started

PRACTICE Macros

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

Macro names in PRACTICE always start with an ampersand sign ('&'), followed by a sequence of letters (a-z, A-Z), numbers (0-9) and the underscore sign ('_'). The first character after the **&** sign must not be a number. Macro names are case sensitive, so &a is different from &A.

Declare a Macro

Empty PRACTICE macros can be declared 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.

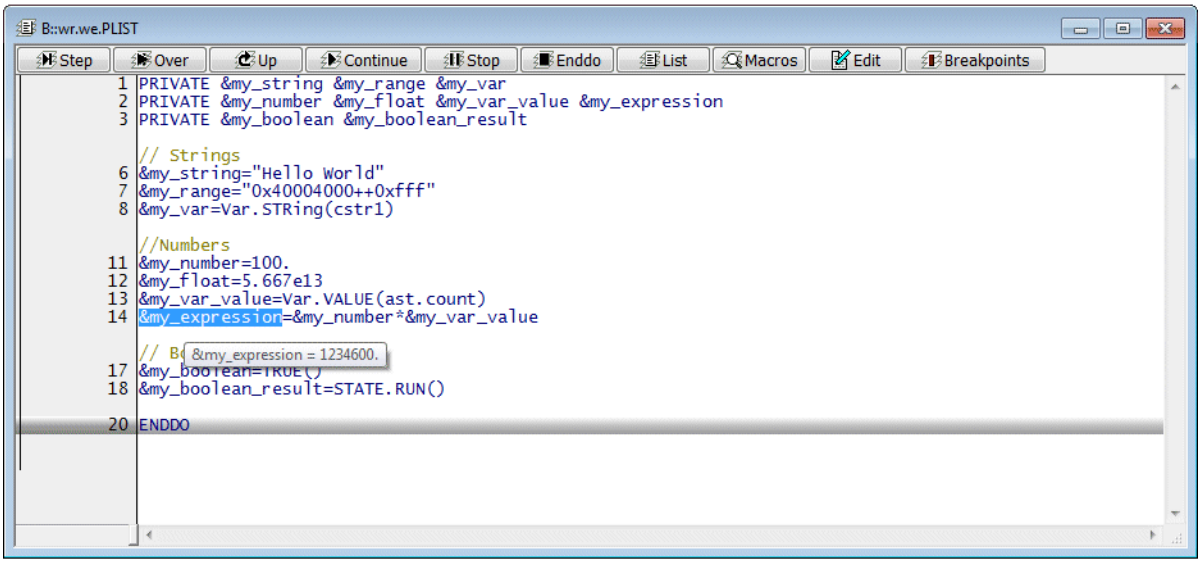
```
// Script test_func_param.cmm

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 implicitly declare a new **LOCAL** macro when an assignment is done and 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 declared 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.

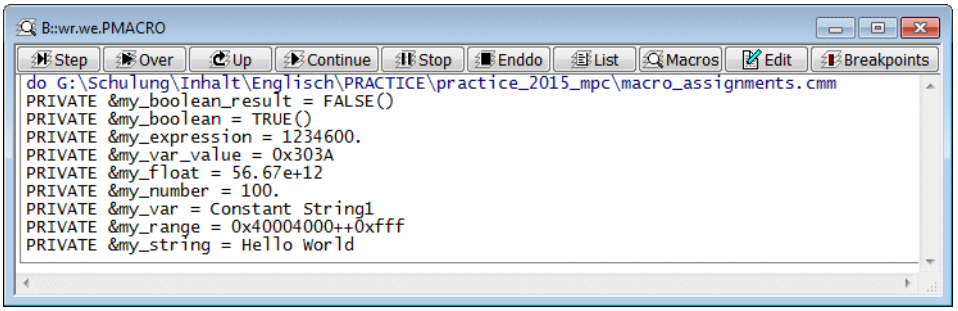
```
// Script

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.



The PRACTICE stack provide an overview for all macros.

The keyword PRIVATE is used to identify **PRIVATE** macros.



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.

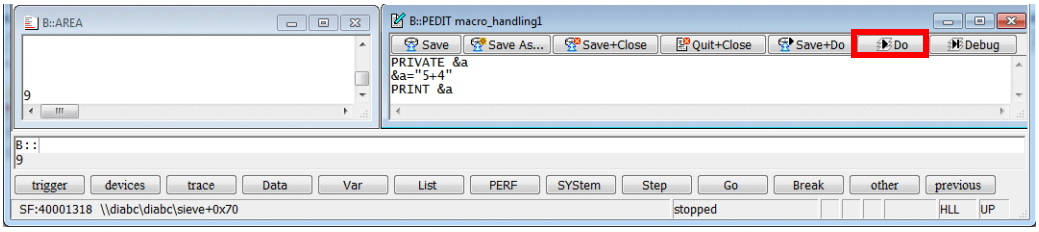
PRINT {*<data>*}	PRINT specified *<data>* to TRACE32 message line and to TRACE32 message area.

Example 1

```
PRIVATE &a

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

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

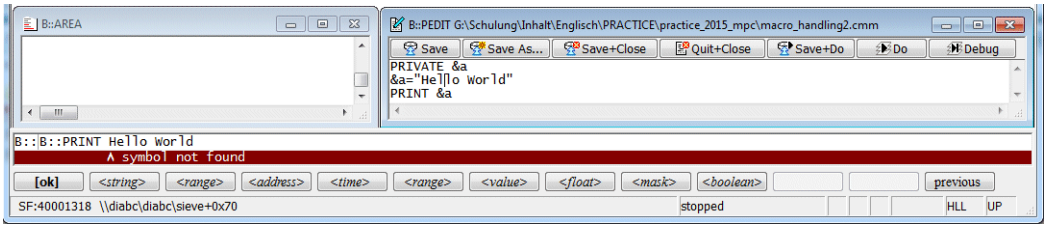
1. Replace all macros with their corresponding character sequences. So:
   `PRINT &a -> PRINT 5+4`
2. Evaluate all expressions
   `PRINT 5+4 -> PRINT 9`
3. Execute the resulting command. So it will execute **`PRINT 9`**.



Example 2

```
PRIVATE &a

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

This example will generate an error.



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`

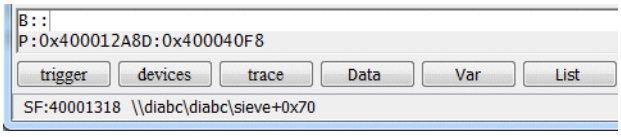
2. Evaluate expressions
   `PRINT Hello World` -> **error**

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**.

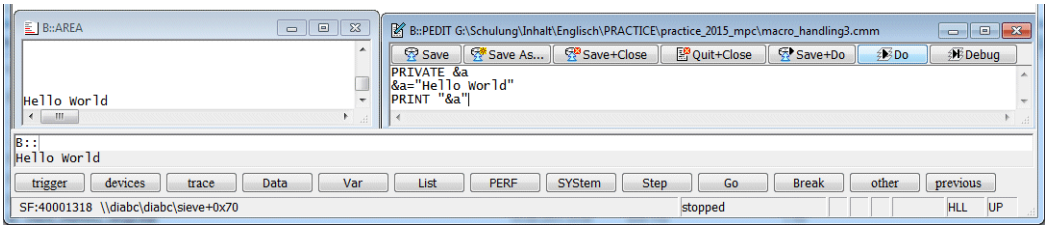
If by pure accident there are debug symbols called **`Hello`** and **`World`** the addresses of these symbols will be printed.



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 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"
```



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.

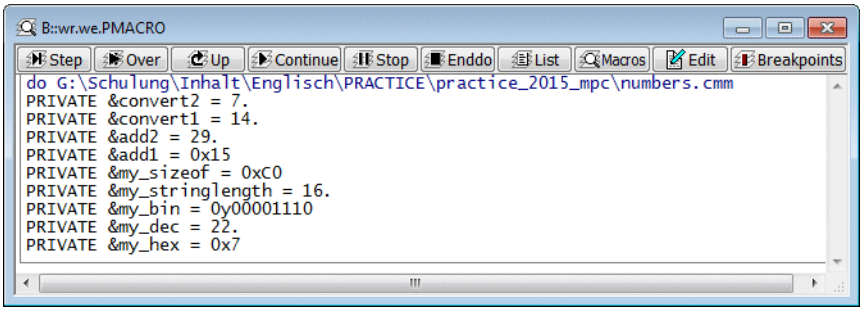
String composing example:

```
// Script string_example.cmm

PRIVATE &drive &architecture &demo_directory
&drive="C:"
&architecture="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.

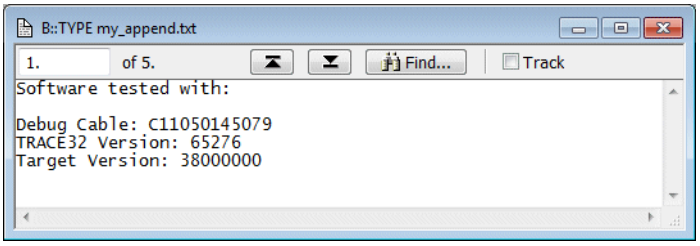


STRing.LENgth(*<string>***)**	Returns the length of the *<string>* as a decimal number.
sYmbol.SIZEOF(*<symbol>***)**	Returns the size occupied by the specified debug *<symbol>* (e.g. function, variable, module) as a hex. number.

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:



TRACE32 command

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

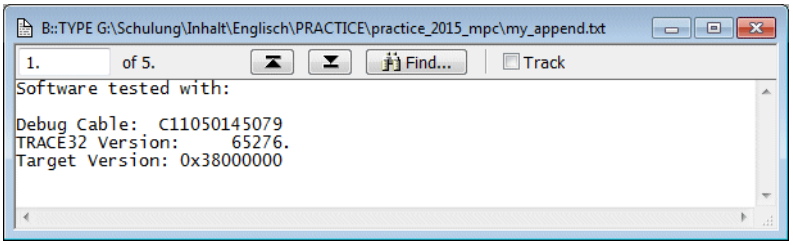
TRACE32 functions

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))
```

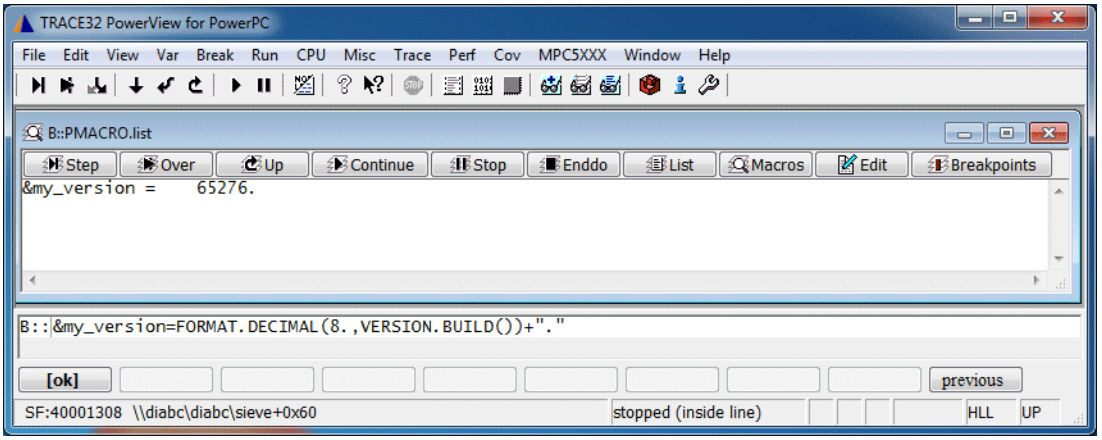
May result for example in:



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.

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



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 or in 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

Run Through Program and Generate a Test Report

Task of part 1 of the script: Start the target 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>* has 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(),0)
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
  ENDDO
)
```

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

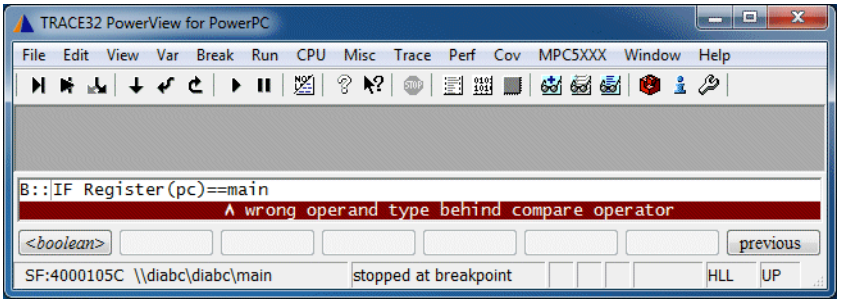
TRACE32 function

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

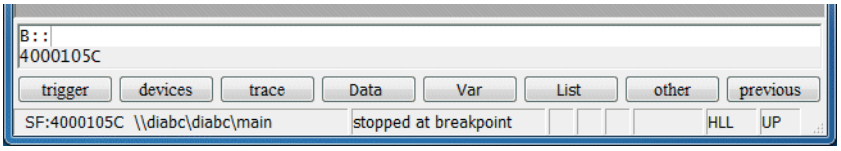
Addresses in TRACE32

Why is the following function needed?

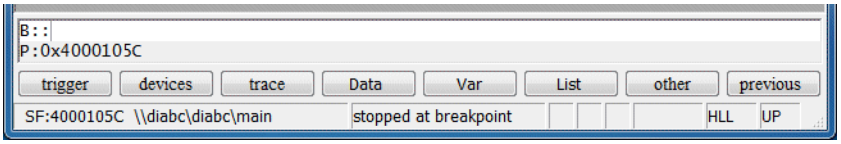
```
ADDRESS.OFFSET(main)
```



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



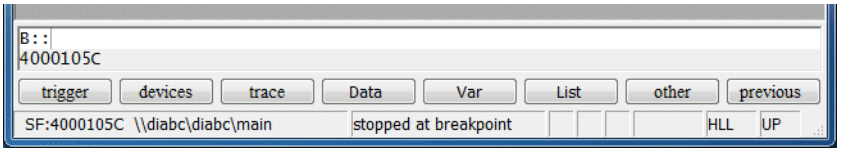

```
PRINT main                              // print the address of main
```



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)
```



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(),0)
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
  ENDDO
)
```

PRACTICE commands:

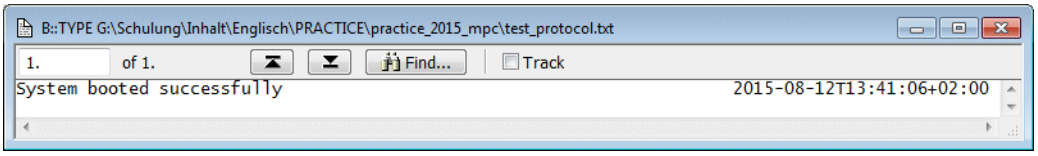
APPEND *<filename>* {*<data>*}	Append data to content of file *<filename>.*
ENDDO	A script ends with its last command or with the ENDDO command.

TRACE32 functions:

FORMAT.STRing(*<string>,<width>,<filling_character>***)**	Formats *<string>* to the specified *<width>.* If the length of *<string>* is shorter the *<width>*, the *<filling_character>* is appended.
DATE.UnixTime()	Returns the time in UNIX format and that is seconds passed since Jan 1st 1970.
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffset())	Format Unix time according to ISO 8601

```
// 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:

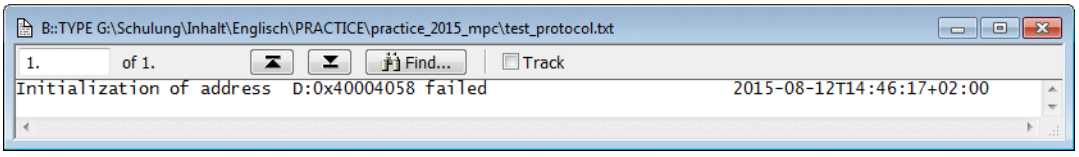


Check Contents of Addresses

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:



TRACE32 function

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

<address> requires an access class.

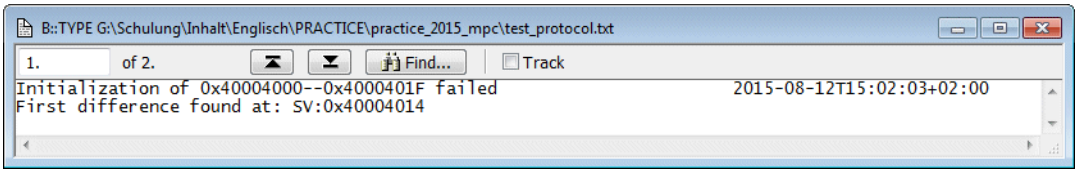
```
Data.Long(D:0x40004058)              // D: indicates the generic access
                                     // class Data

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:



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

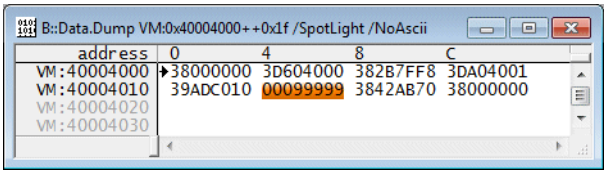
FOUND()	Returns TRUE if a difference was found.
TRACK.ADDRESS()	Returns the address of the first difference.

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 contains the instructions between the 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:



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.

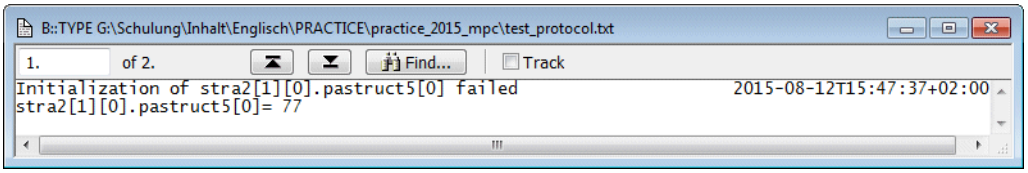
SCREEN.display	Advise TRACE32 to update the screen now.

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:



PRACTICE command

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

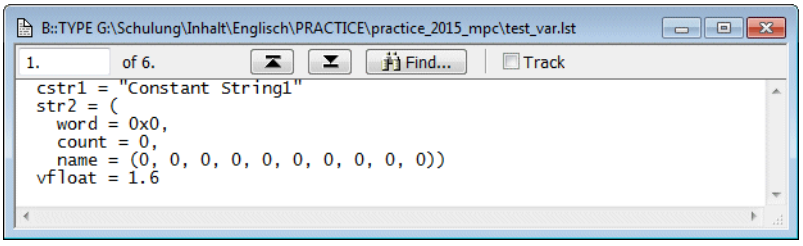
PRACTICE function

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

Record Formatted Variables

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
…
```



TRACE32 commands

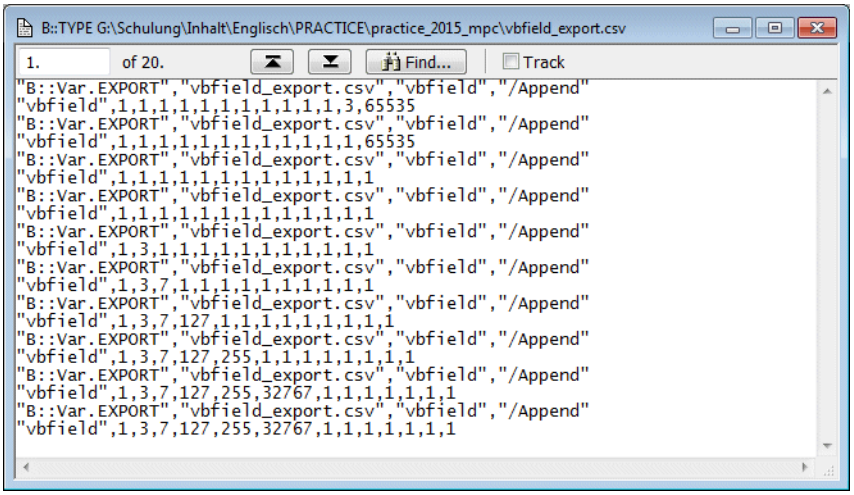
PRinTer.FileType *<format>*	Specify output *<format>* for output file.
PRinTer.Open *<filename>*	Open file *<filename>* for outputs.
PRinTer.CLOSE	Close open output file.
WinPrint.*<command>*	Redirect the *<command>* output to the specified file.
WinPos ,,,,,0	By the default the TRACE32 *<command>* and 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:



PRACTICE command

// 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**

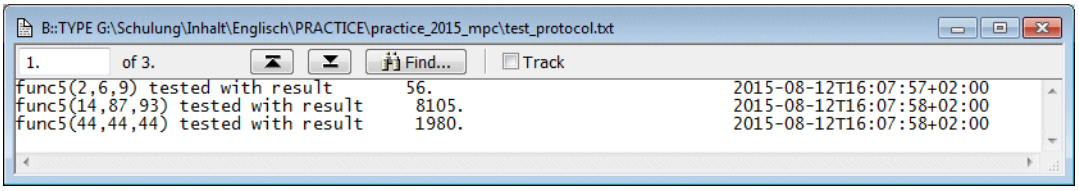
```
;Example 1
RePeaT 100. PRINT "X"                          ;Print X 100 times

;Example 2
RePeaT 50.                                     ;Print a * at the end of
(                                              ;the previous line then
    PRINT %CONTinue "*"                        ;wait 100 ms. Do this 50
    WAIT 100.ms                                ;times.
)

;Example 3
LOCAL &code
AREA.Create A000 , 2100.                       ;Change the number of
AREA.view A000                                 ;lines in the AREA window
                                               ;to 2100
OPEN #1 "~~/t32.men" /Read
RePeaT                                         ;Read until the end of
(                                              ;file
    READ #1 %LINE &code
    PRINT "&code"
)
WHILE EOF()==FALSE()
CLOSE #1
```

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:



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

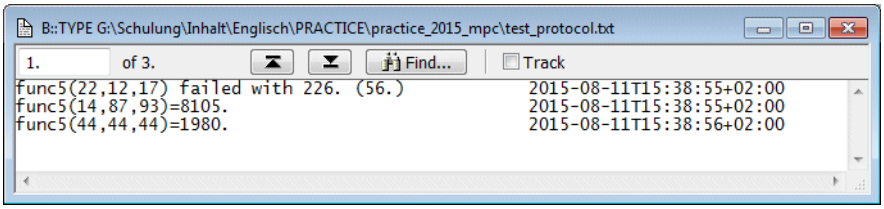
Task of script: Test functions, but provide function name, parameters and expected result by a parameter file. Generate a test protocol.

```
// 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:



PRACTICE command

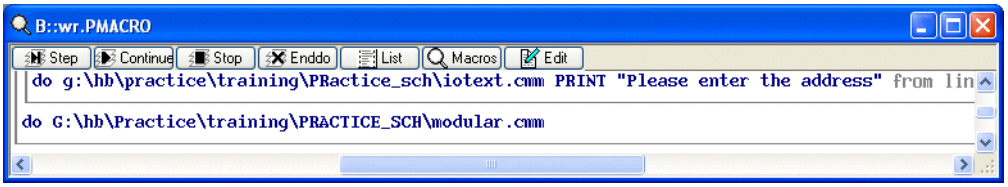
GOSUB *<label>*	Call a subroutine. The start of the subroutine is identified by *<label>.* Labels must start in the first column of a line and end with a colon. No preceding white space allowed. Subroutines are usually located after the ENDDO statement.
RETURN	Return from subroutine.
LOCAL *<macro>*	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 here is 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.

Pass Parameters to a PRACTICE Script or to a Subroutine



ENTRY *<parlist>*

The **ENTRY** command can be used to

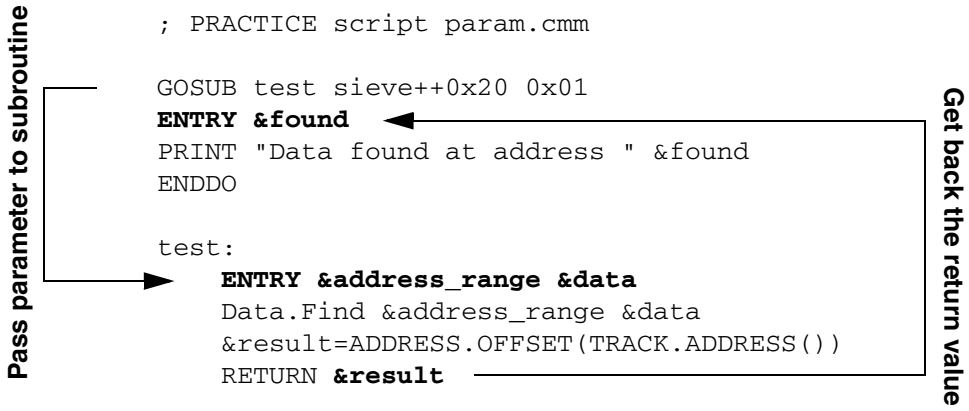
- Pass parameters to a PRACTICE script or to a subroutine
- To return a value from a subroutine

Example 1: Pass parameters to a PRACTICE script

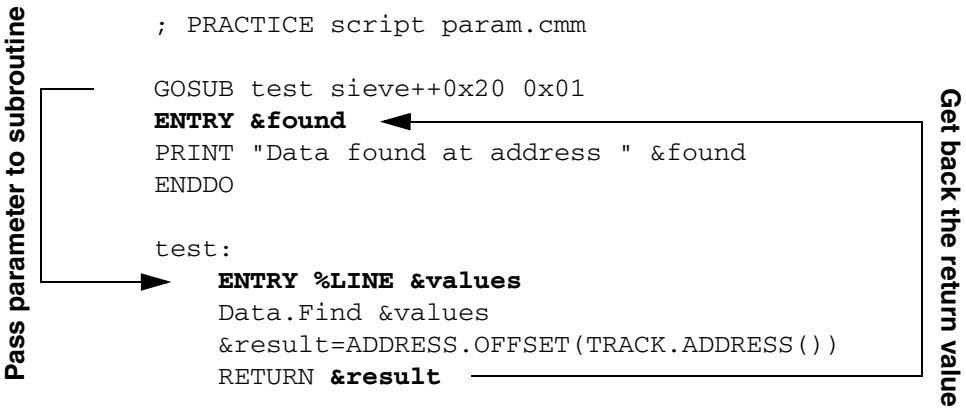
```
; PRACTICE script patch.cmm
; DO patch.cmm 0x1000++0xff 0x01 0x0a

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
```

Data.FIND *<address_range> <data>*	Search for *<data>* in the specified *<address_range>.* TRACE32 functions served: **FOUND()** **TRACK.ADDRESS()**
FOUND()	Returns a boolean value to denote the results of the last **DATA.FIND** command.
TRACK.ADDRESS()	Returns the address of the last successful search.

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

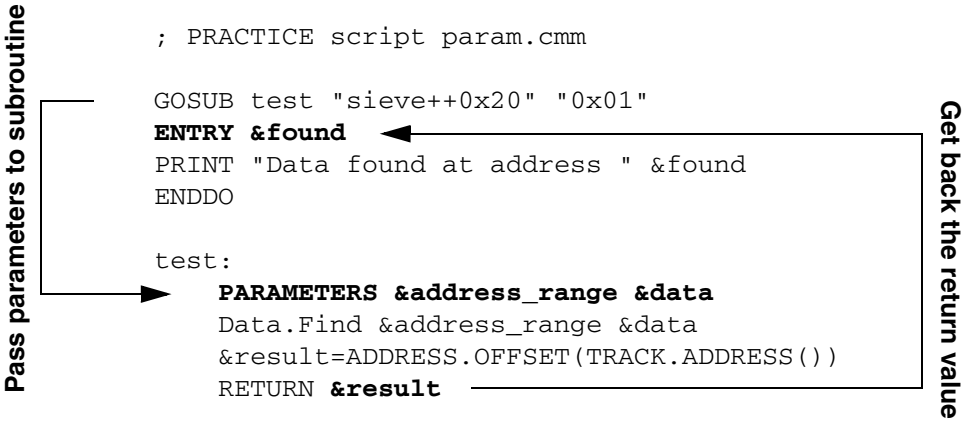


Example 3: Pass arguments as a single parameter to a subroutine and get back the return value



Alternatively, the function **PARAMETERS** can be used to read parameters passed to a script or subroutine and, if the macros do not already exist in the current scope, assign them to automatically created **PRIVATE** macros. The parameters must be enclosed in quotation marks (").

Example 4: Pass arguments using `PARAMETERS` to a subroutine and get back the return value



Example 5: Passing arguments via the command line. If the values for **&data_old** and **&data_new** are not provided default values are assigned.

```
; PRACTICE script patch_not_all.cmm
; Call with:
; DO patch_not_all.cmm "<address_range>" "<data_old>" "<data_new>"


PARAMETERS &address_range &data_old &data_new

IF "&data_old"==""
(
   &data_old=0x00
)

IF "&data_new"==""
(
   &data_new=0xFF
)

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
```

Passing parameters into scripts like this means that the user must know the order in which each parameter must be specified. A more flexible approach is to use the function **STRing.SCANAndExtract()**.

STRing.SCANAndExtract("*<source>*", "*<search>*", "*<default>*")	The *<source>* string is searched for the pattern *<search>* and if it is found the character sequence after and up to the next blank space is assigned to the PRACTICE macro. If the *<search>* string is not found the value *<default>* is assigned instead.

.

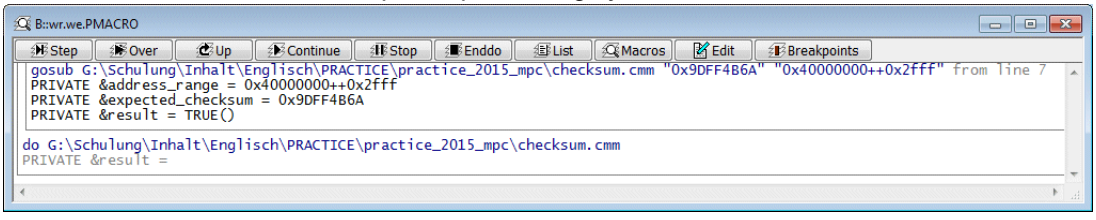
```
;Call this script with:
; do arg_test.cmm ADDRESS=<address> DATA=<data> COUNT=<count>

;Read all command line arguments as a single line
ENTRY %LINE &p

;Open an AREA window and clear it for displaying the results
AREA.view
AREA.CLEAR

;If the ADDRESS argument is not provided, print a usage message
IF STRing.SCAN("&p","ADDRESS=",0)==-1
(
    GOSUB print_args
)

;Extract the arguments into the correct PRACTICE variables, returning the
;default value if the user does not specify something.
&addr=STRing.SCANAndExtract(STRING.UPpeR("&p"), "ADDRESS=", "0x12000")
&dvalue=STRing.SCANAndExtract(STRING.UPpeR("&p"),"DATA=", "0x00")
&count=STRing.SCANAndExtract(STRING.UPpeR("&p"),"COUNT=","5.")

PRINT "Extracted Arguments:"
PRINT "ADDRESS = &addr"
PRINT "DATA    = &dvalue"
PRINT "COUNT   = &count"
ENDDO

print_args:
   PRINT "Usage:"
   PRINT
   PRINT "do arg_test.cmm ADDRESS=<address> DATA=<data> COUNT=<count>"
   PRINT "Where:"
   PRINT "  <address>   is the address to write <data> to."
   PRINT "  <count>  is the number of times to write the <data> value."
   PRINT
   PRINT "Script will complete using default values."
   PRINT
RETURN
```

STRing.UPpeR(*<string>***)** Returns *<string>* in upper case.

Example 6: Call a subroutine to verify a checksum for a given range.

```
; PRACTICE script checksum.cmm

PRIVATE &result

AREA.view
AREA.CLEAR
DATA.LOAD.BINARY checksum.bin 0x40000000

GOSUB verify_checksum "0x9DFF4B6A" "0x40000000++0x2fff"
RETURNVALUES &result

IF &result
   PRINT %COLOR.GREEN "First checksum verification passed"
ELSE
   PRINT %COLOR.RED "First checksum verification failed"
WAIT 1.s

; Modify memory
Data.Set 0x40000100 %Long 0x12345678

GOSUB verify_checksum "0x9DFF4B6A" "0x40000000++0x2fff"
RETURNVALUES &result

IF &result
   PRINT %COLOR.GREEN "Second checksum verification passed"
ELSE
   PRINT %COLOR.RED "Second checksum verification failed"

ENDDO


verify_checksum:
(
   PRIVATE &result
   PARAMETERS &expected_checksum &address_range

   Data.SUM &address_range /CRC32
   &result=(Data.SUM()==&expected_checksum)
   RETURN "&result"
)
```

PRIVATE macros that are out scope are printed in gray in the PRACTICE stack.



TRACE32 commands

AREA.view	Display TRACE32 message area.
AREA.CLEAR	Clear TRACE32 message area.
Data.Set *<address> <value>*	Write *<value>* to *<address>.*
Data.SUM *<address_range>* **/CRC32**	Calculate CRC32 checksum for specified *<address_range>.*

PRACTICE command

PRINT %COLOR.*<color>* {*<data>*}	Print the specified data in the specified color to the TRACE32 message area.

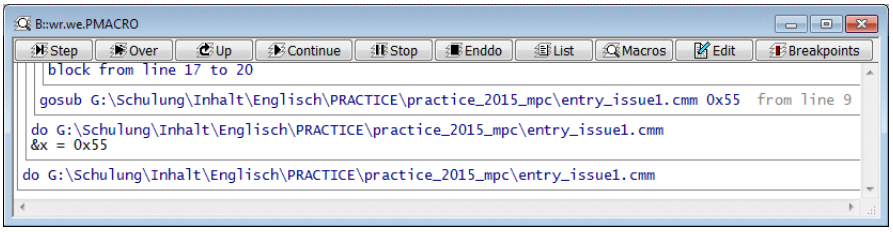
TRACE32 function

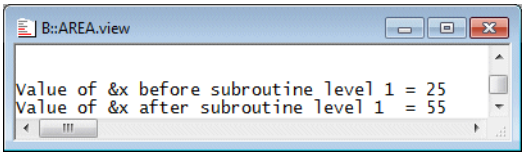
Data.SUM()	Returns checksum calculated by the **Data.SUM** command as hex. number.

Using the PRACTICE commands **PARAMETERS**/**RETURNVALUES** solve the following issues known for the **ENTRY** command.

1. Called script/subroutine may unintentionally overwrite LOCAL macros of the caller(s), if the macros used with the ENTRY command are not explicitly created.

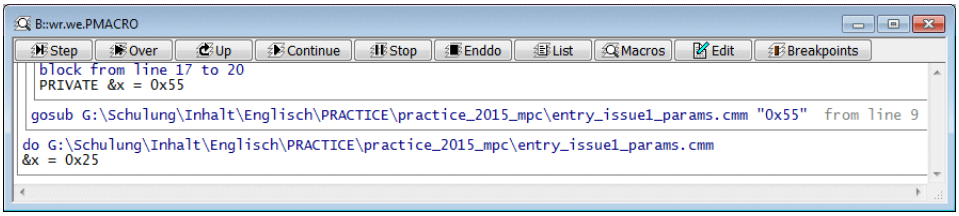
```
// Script entry_issue1.cmm

AREA.view
AREA.CLEAR

LOCAL &x
&x=0x25

PRINT "Value of &"+"x before subroutine level 1 = " &x

GOSUB level1 0x55

PRINT "Value of &"+"x after subroutine level 1  = " &x

ENDDO


level1:
(
  ENTRY &x
  RETURN
)
```

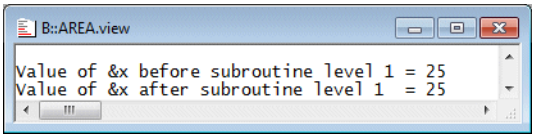




```
// Script entry_issue1_params.cmm

AREA.view
AREA.CLEAR

LOCAL &x
&x=0x25

PRINT "Value of &"+"x before subroutine level 1 = " &x

GOSUB level1 "0x55"

PRINT "Value of &"+"x after subroutine level 1  = " &x

ENDDO


level1:
(
  PARAMETERS &x
  RETURN
)
```

Notice that the **LOCAL** macros of the caller(s) are out of scope for the **PARAMETERS** command. So a new **PRIVATE** macro &x is created.

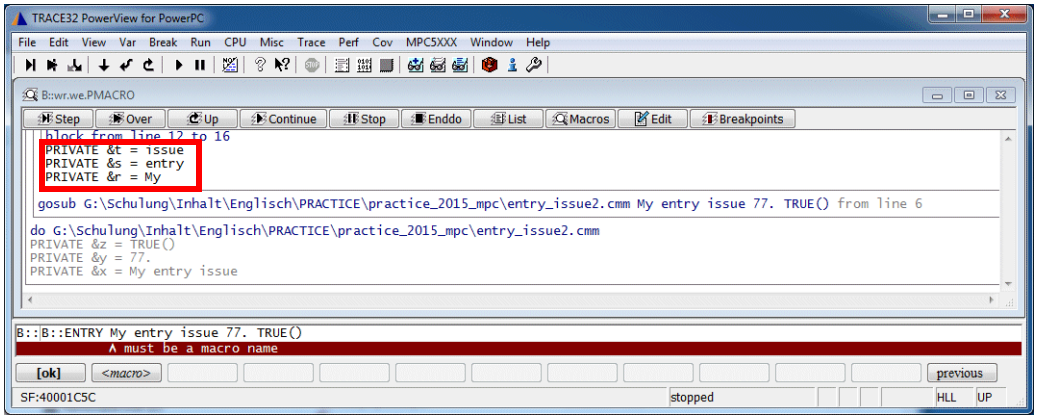




2. Whitespace as parameter delimiter.

The **ENTRY** command regards whitespace as a parameter delimiter. If one of the parameters for the called script/subroutine is a string containing whitespaces and it is not quoted, the script is stopped by an error, because not enough macros are provided to take all parameters.

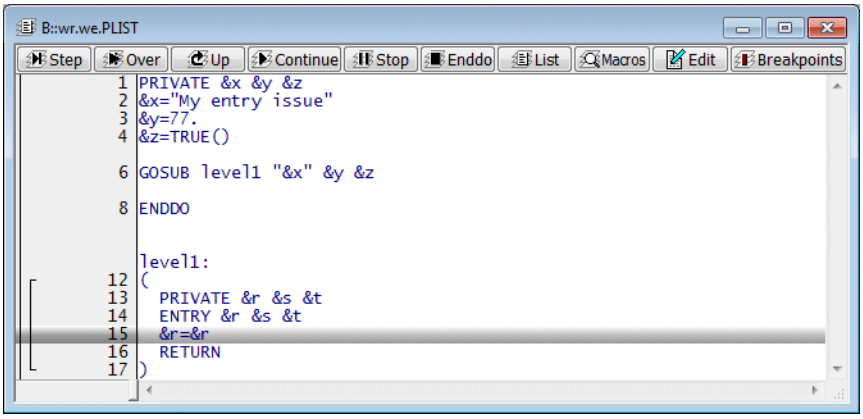
```
// Script entry_issue2.cmm

PRIVATE &x &y &z
&x="My entry issue"
&y=77.
&z=TRUE()

GOSUB level1 &x &y &z

ENDDO


level1:
(
  PRIVATE &r &s &t
  ENTRY &r &s &t
  RETURN
)
```

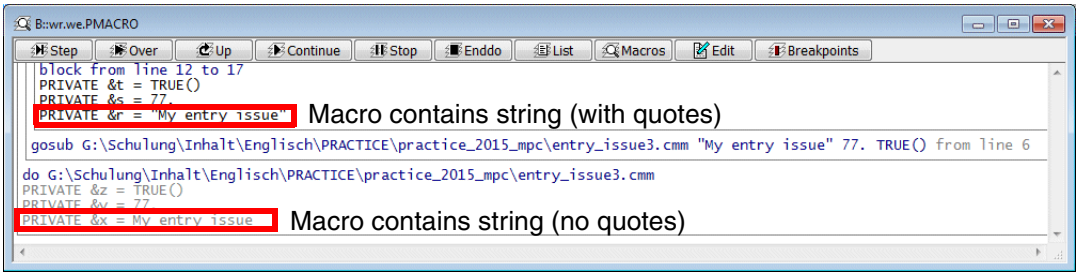


3. The quote issue.

If the string containing whitespaces is quoted, the quotes become part of the string if the **ENTRY** command is used.

```
// Script entry_issue3.cmm

PRIVATE &x &y &z
&x="My entry issue"
&y=77.
&z=TRUE()

GOSUB level1 "&x" &y &z

ENDDO


level1:
(
  PRIVATE &r &s &t
  ENTRY &r &s &t
  &r=&r                    // Removes quotes from string
  RETURN
)
```

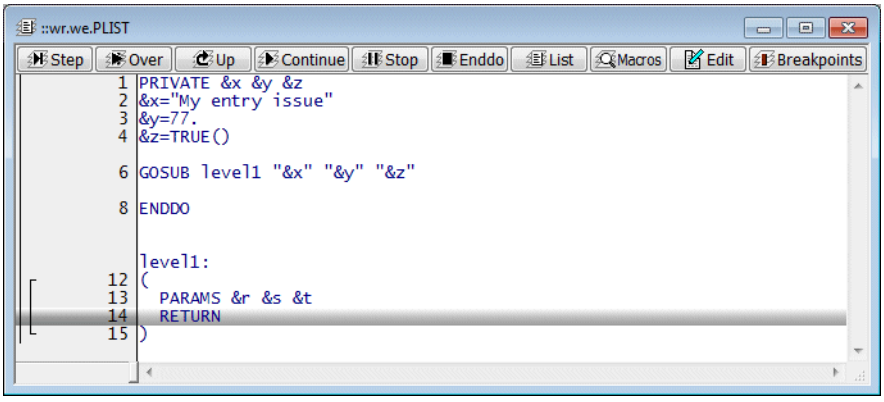


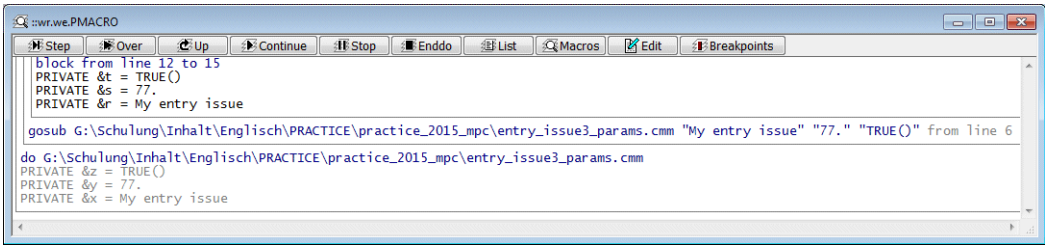



```
// Trick to remove of the quotes
&r=&r
```

The **PARAMETERS** command solves both issues.

```
// Script entry_issue3_params.cmm

PRIVATE &x &y &z
&x="My entry issue"
&y=77.
&z=TRUE()

GOSUB level1 "&x" "&y" "&z"

ENDDO


level1:
(
  PARAMETERS &r &s &t
  RETURN
)
```





Operating System Interaction

PRACTICE can interact with the Operating System on the host computer. A well written PRACTICE script can hide the differences in underlying Operating System and present a unified experience to the user.

Operating System Detection

Sometimes differences in the underlying host Operating System require the PRACTICE script to be a bit more flexible in order to be truly portable. The function **OS.VERSION(0)** can be used to detect the host Operating System.

```
;Printout host OS
IF OS.VERSION(0)>=0x50
  PRINT "Unkown OS"
ELSE IF OS.VERSION(0)>=0x40
  PRINT "Mac OS X"
ELSE IF OS.VERSION(0)>=0x30
  PRINT "HP-UX"
ELSE IF OS.VERSION(0)>=0x10
  PRINT "Linux"
ELSE
  PRINT "MS Windows"
ENDDO
```

```
;Open a serial terminal - host dependent code
TERM.RESet
IF OS.VERSION(0)<0x10           ;MS Windows
(
    TERM.METHOD COM COM1 9600. 8 NONE 1STOP NONE
)
ELSE                            ;Linux?
(
    TERM.METHOD COM /dev/ttyS0 9600. 8 NONE 1STOP NONE
)
TERM.SCROLL ON
TERM.view
ENDDO
```

Printing Results

Results can be sent to the system clipboard, a file or a connected printer. Physical printers must be configured correctly in the TRACE32 configuration file (usually `~~/config.t32`), although Windows users will not need to do anything here.

To print results:

1. Select a printer using **PRinTer.select** (for manual selection via GUI), **PRinTer.FILE**.
2. Optionally format the output: **PRinTer.FileType** | **PRinTer.ClipBoard** / **PRinTer.SIZE** /**PRinTer.OFFSET**
3. Open a connection to the printer with **PRinTer.OPEN**. Use **PRinTer.EXPORT** to generate CSV data.
4. Re-direct the output from the command to the chosen printer using **WinPrint**.*<command>*
5. Close the printer connection with **PRinTer.CLOSE**

For a more detailed explanation of the commands and options, refer to **"PRinTer"** in PowerView Command Reference, page 261 (ide_ref.pdf).

```
; Example 1 - save a memory dump to a file
LOCAL &file
; Get a temporary file from the OS
&file=OS.TMPFILE()

PRinTer.FILE "&file"
PRinTer.FileType ASCIIE
WinPrint.Data.dump 0x2200--0x22FF
&file="&file"+".lst"
; Display the results
TYPE "&file"
ENDDO
```

```
; Example 2 - Save two memory dumps as CSV files for viewing in Excel
LOCAL &file
&file=OS.TMPFILE()

; Adding a numerical suffix to the file name will allow
; PRinTer.EXPORT to automatically increment the filename
; each time it is used. Otherwise it over-writes the existing file.
&file="&file"+"-1"

PRinTer.EXPORT.default "&file"

WinPrint.Data.dump 0x2200--0x22FF /Word
WinPrint.Data.dump 0x2300--0x23FF /Byte

&file="&file"+".lst"
TYPE "&file"
&file=STRing.Replace("&file","-1.lst","-2.lst",0)
TYPE "&file"
ENDDO
```

```
; Example 3 - Save the same memory dump as ASCII and CSV to
; the same file
LOCAL &file
&file=OS.TMPFILE()

; Set format to CSV initially
PRinTer.FileType CSV
PRinTer.OPEN "&file"

WinPrint.Data.dump 0x2200--0x22FF /Word

; Close the file, set the format to ASCIIE, then re-open the
; file to append new data
PRinTer.CLOSE
PRinTer.FileType ASCIIE
PRinTer.OPEN "&file" /Append

WinPrint.Data.dump 0x2200--0x22FF /Word

PRinTer.CLOSE

&file="&file"+".lst"
TYPE "&file"
ENDDO
```

Accessing Environment Variables

Environment variables can be accessed from a PRACTICE script by using the function **OS.ENV("<env_var>")**. The value of the variable is returned and can be assigned to a PRACTICE macro.

```
; This example will only work on a Windows host
LOCAL &username
&username=OS.ENV("USERNAME")
AREA.view
PRINT "Hello, &username"
ENDDO
```

A more extensive example which will run on Windows, Linux or MacOS can be found at `~~/demo/practice/my_environment.cmm`.

Running a Command

The OS.* family of commands can be used to launch programs on the host Operating System from within a PRACTICE scripts. A brief overview is given here but more information can be found at **"OS"** in PowerView Command Reference, page 241 (ide_ref.pdf).

OS.Area *<cmdline>*	Execute a command on the host. The results will be displayed in the AREA window. the script will block until the command has completed.
OS.Command *<cmdline>*	Execute multiple commands on the host (use host native piping). PRACTICE macros can be inserted into the command line. On Windows hosts, the first argument must be the command interpreter (CMD.EXE). On Windows hosts the PRACTICE script does not block. On other hosts use '&' to force non-blocking behavior.
OS.Hidden *<cmdline>*	Execute a command; the output is discarded. PRACTICE script blocks until the command completes.
OS.screen *<cmdline>*	Open a host shell and execute the command line. This is non-blocking.
OS.Window *<cmdline>*	Execute a host command and all output will be re-directed to a new TRACE32 window. This is non-interactive and the PRACTICE script blocks until the command has completed.

```
; List all *.cmm files in the current AREA window
; For Windows hosts
OS.Area dir *.cmm
; For Linux/MacOS hosts
OS.Area ls -l *.cmm

; List all *.cmm files in a new TRACE32 window (OS.Window)
; For Windows hosts
OS.Window dir *.cmm
; For Linux/MacOS hosts
OS.Window ls -l *.cmm

; List all *.cmm files to a file in the user's home directory
; For Windows hosts
OS.Command CMD /C dir *.cmm > %USERPROFILE%/tmpfile.lst
; For Linux/MacOS hosts
OS.Command ls -l *.cmm > $HOME/tmpfile.lst
```

File Manipulation

A PRACTICE script can manipulate files in the host Operating System file system (display, copy, delete, etc.). A number of commands are available and it is recommended to use these instead of the underlying Operating Systems' native commands to make scripts more portable. More information can be found at **"File and Folder Operations"** in PowerView User's Guide, page 77 (ide_user.pdf).

In addition, PRACTICE supports 9 file handles (#1 through #9) that are used to open, close, read and write files.

- Open file

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

- Close file

  CLOSE #*<buffer>*

- Read data from an open file

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

- Write data to an open file

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

```
; Assume file "file.txt" contains
; Hello and Welcome

; Read each word
LOCAL &arg1 &arg2 &arg3 &line
OPEN #1 "file.txt" /Read
READ #1 &arg1 &arg2 &arg3
CLOSE #1

; Print them in reverse order
AREA.view
PRINT "&arg3 &arg2 &arg1"

; Read the entire line
OPEN #1 "file.txt" /Read
READ #1 %LINE &line
CLOSE #1
PRINT "Line Reads: &line"
ENDDO
```

Data can be written using **WRITE** or **WRITEB** (for binary data).

```
; Example of writing text data to a file in different styles
LOCAL &fname &sname &file

DIALOG.view
(
   HEADER "Write Test"
   POS 0. 0. 30. 4.
   BOX "Please Enter your details"
   POS 1. 1. 10. 1.
   TEXT "First name:"
   POS 1. 2. 10. 1.
   TEXT "Last name:"
   POS 12. 1. 17. 1.
F: DEFEDIT "" ""
   POS 12. 2. 17. 1.
S: EDIT "" ""
   POS 24. 4. 6. 1.
   DEFBUTTON "OK" "CONTinue"
)
STOP
&fname=DIALOG.STRING(F)
&sname=DIALOG.STRING(S)
DIALOG.END

&file=OS.TMPFILE()
OPEN #1 "&file" /CREATE
; Write first line
WRITE #1 "File saved for &fname &sname"

; Write second line which should be identical to the first
WRITE #1 "File saved for "+"&fname"+" "+"&sname"

; Write third line which will be identical to the other two
; The %CONTinue allows the current line to be on the same line as the
; previous write.
WRITE #1 "File saved for " "&fname"
WRITE #1 %CONTinue " " "&sname"

; Close the file
CLOSE #1
TYPE &file
ENDDO
```

```
; Example of writing binary data to a file
LOCAL &file

&file=OS.TMPFILE()
OPEN #2 "&file" /Binary /Create
WRITEB #2 0x48 0x65 0x6C 0x6C 0x6F
WRITEB #2 0x20 0x57 0x6F 0x72 0x6C 0x64
CLOSE #2

TYPE &file
ENDDO
```

These techniques can be combined with the printing to a file method from earlier to extract and parse information from a target system. The example below shows how to build a list of all the running threads in a FreeRTOS based system and display them in a custom dialog.

```
LOCAL &tasklist &tmpfile &line &tmpline &tfile
&tasklist=""
&tmpfile=OS.TMPFILE()
&line=""

; Print task list window contents to a temporary file
PRinTer.FILE "&tmpfile"
PRinTer.FileType ASCIIE
WinPrint.TASK.TaskList
&tfile="&tmpfile"+".lst"

OPEN #1 "&tfile"
; Read first two lines and throw them away
; These are the header and column titles
READ #1 %LINE &line
READ #1 %LINE &line

WHILE !FILE.EOF(1)
(
  READ #1 %LINE &line
  IF "&line"!=""
(
  ; Extract only the task name part of the line
    &tmpline=STRing.MID("&line",10.,16.)
    &tasklist="&tasklist"+"&tmpline"+","
  )
)
CLOSE #1
DIALOG.view
(
     HEADER "FreeRTOS Task List"
     POS 0. 1. 25. 10.
TSK: LISTBOX "" ""
     POS 12. 12. 10. 1.
     DEFBUTTON "Close" "CONTinue"
)
DIALOG.Set TSK "" "&tasklist"
STOP

DIALOG.END
ENDDO
```

Time and Date Functions

The date and time can be retrieved using the **DATE.DATE()** and **DATE.TIME()** functions respectively. More information about date and time functions can be found at **"DATE Functions"** in PowerView Function Reference, page 37 (ide_func.pdf).

```
AREA.view
PRINT DATE.DATE()
PRINT DATE.TIME()
ENDDO
```

These can be used to add timestamps to reports or log files or to include the time and date in log file names.

```
LOCAL &date &time &d &m &y &logfile
&date=DATE.DATE()
&time=DATE.TIME()
&d=DATE.DAY()
&m=DATE.MONTH()
&y=DATE.YEAR()

; Remove '.' and ':' characters
&d=STRing.Replace("&d",".","",0)
&m=STRing.Replace("&m",".","",0)
&y=STRing.Replace("&y",".","",0)
&time=STRing.Replace("&time",":","_",0)

; Create logfile name
&logfile="log_"+"&d"+"_"+"&m"+"_"+"&y"+"_"+"&time"+".log"

; Check to see if logdir is valid or not.
; If not, create it.
IF !OS.DIR("logdir")
   MKDIR "logdir"

PRINT "Creating new log file: &logfile"

OPEN  #1 "logdir/&logfile" /Create
WRITE #1 "LOG File created on &date at &time"
CLOSE #1

TYPE "logdir/&logfile"
ENDDO
```

I/O Commands

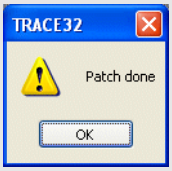
Output Command

PRINT *<format> <parlist>*

DIALOG.OK *<message>*

```
PRINT "FLASH programmed successfully"

PRINT %ERROR "FLASH programming failed"
```

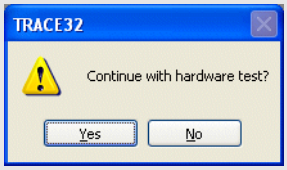
```
DIALOG.OK "Patch done"
```



Input Command

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

```
INKEY                                                   ; Wait for any key

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


; PRACTICE script dialog.cmm
DIALOG.YESNO "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
```

I/O via the AREA Window

An I/O window is needed for PRACTICE inputs and outputs. This is handled by an **AREA** window.

To 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>] [<window_name>] [<state>]
[<header>]
```

4. Display **AREA** window

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

To 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 window.

```
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
```

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)
)
…
```

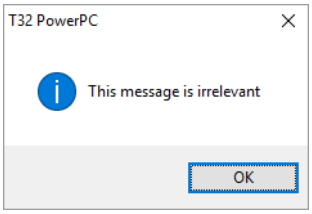
```
ON POWERUP GOTO startup

IF !(STATE.POWER())
     STOP

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

ENDDO
```

Simple Dialogs

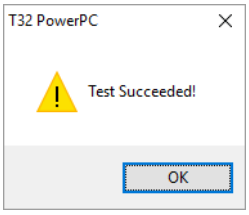
TRACE32 provides a number of simple dialogs which can be used to inform the user of something or to prompt them to make a simple choice.

DIALOG.MESSAGE "*<text>*"	Creates a standard dialog box with an information icon and the user text displayed. The dialog closes when the user clicks the **OK** button.
DIALOG.OK "*<text>*"	Creates a standard dialog box with an exclamation icon and the user provided text. The dialog closes when the user clicks the **OK** button.
DIALOG.YESNO "*<text>*"	Creates a standard dialog box with a question mark icon and the user provided text. The user is presented with two buttons. Clicking **YES** returns a TRUE value to the script whilst clicking **NO** returns a FALSE value to the script. Either button will close the dialog.

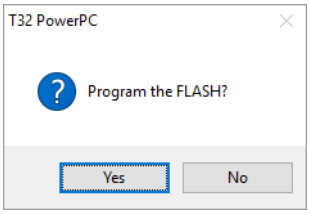
```
DIALOG.MESSAGE "This message is irrelevant"
```




```
DIALOG.OK "Test Succeeded!"
```



```
LOCAL &result
DIALOG.YESNO "Program the FLASH?"
ENTRY &result
if &result
    GOSUB prog_flash
ENDDO
```



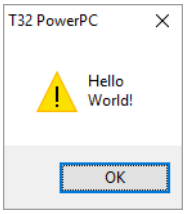
Text can be split across multiple lines in these simple dialogs. The list below shows the methods of achieving this.

```
DIALOG.OK "Hello"+CONVert.CHAR(0x0D)+"World!"
```

```
DIALOG.OK "Hello"+CONVert.CHAR(0x0A)+"World!"
```

```
DIALOG.OK "Hello"+CONVert.CHAR(0x0A0D)+"World!"
```

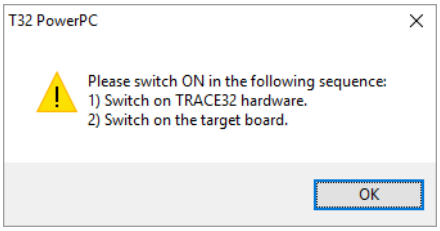
```
DIALOG.OK "Hello" "World"
```



Three of these use the **CONVert.CHAR**() function to insert carriage return (`0x0D`, `'\r'`) and/or line feed (`0x0A`, `'\n'`) characters into the string. The fourth version simply presents two strings and TRACE32 will put one per line.

Using the technique for splitting PRACTICE lines can help break up dialogs into something more readable.

```
DIALOG.OK "Please switch ON in the following sequence:" \
"1) Switch on TRACE32 hardware," \
"2) Switch on the target board."
```



Complex dialogs can be created in PRACTICE scripts. These allow users to develop custom interfaces for their scripts. Dialogs can be created in one of two ways. The first requires putting the dialog code into a separate file with a **.dlg** extension and calling it with the **DIALOG.view** command.

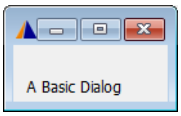
DIALOG.view *<filename>*	Creates a dialog based upon the code in the dialog file, *<filename>*.

```
; File basicdialog.dlg

POS 1. 1. 10.
TEXT "A Basic Dialog"


; File dialogscript.cmm
DIALOG.view basicdialog.dlg
SCREEN.WAIT   //update screen
WAIT 3.s
DIALOG.END
ENDDO
```

The second method includes the dialog code within the script itself.

```
; Example code to create a simple dialog

DIALOG.view
(
    POS 1. 1. 10.
    TEXT "A Basic Dialog"
)
SCREEN.WAIT   //update screen
WAIT 3.s
DIALOG.END
ENDDO
```

Both examples produce the same result. Dialogs automatically size themselves to encompass all of the controls placed within it.

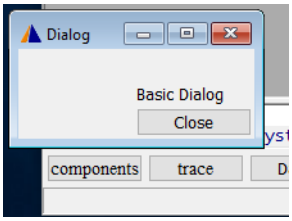


DIALOG.END	Closes the currently active dialog window.

By default dialog windows are contained within the parent window of TRACE32 unless you have specified a non-MDI style interface. Dialogs can be created as free-floating windows by using the **WinExt** command.

WinExt.<*command*>	Pre-command for creating an external window.

```
; Example of non-MDI dialog
WinExt.DIALOG
(
    TEXT "Hello"
    BUTTON "Close" "CONTINUE"
)
STOP
DIALOG.END
ENDDO
```



By using the PRACTICE command **STOP** in this example, the dialog becomes modal. The script will wait until the user (or the script) continues execution. This **CONTinue** comes when the command associated with the **BUTTON** control is executed. This is activated when the user clicks the button.

Adding the prefix **WinResist** will cause the dialog to remain displayed after a call to **WinCLEAR**. Multiple command prefixes can be applied to each dialog.

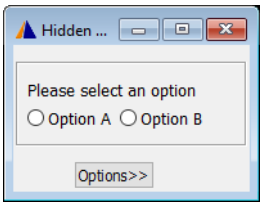
```
; Example of non-MDI dialog which will persist through a WinCLEAR
; operation.
WinResist.WinExt.DIALOG
(
    TEXT "Hello"
    BUTTON "Close" "CONTinue"
)
STOP
DIALOG.END
ENDDO
```

A dialog can be resized using the **WinRESIZE** command. This can be performed dynamically and allows for hidden options.

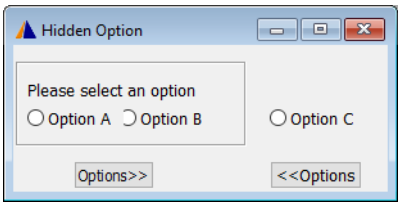
WinRESIZE [*<width>*] [*<height>*] [*<windowname>*]	Resizes the open window that has the name *<windowname>*. If no name is specified, the top most window is resized. A *<windowname>* can be created with the **WinPOS** or **NAME** commands.

```
WinExt.DIALOG
(
     HEADER "Hidden Option"
     POS 0. 0. 20. 4.
     BOX ""
     POS 1. 1. 15. 1.
     TEXT "Please select an option"
     POS 1. 2. 8. 1.
A.A: CHOOSEBOX "Option A" ""
     POS 9. 2. 8. 1.
A.B: CHOOSEBOX "Option B" ""
     POS 22. 2. 8. 1.
A.C: CHOOSEBOX "Option C" ""
     POS 5. 4. 7. 1.
     BUTTON "Options>>"
     (
        WinRESIZE 32. 5.
     )
     POS 22. 4. 8. 1.
     BUTTON "<<Options"
     (
        WinRESIZE 20. 5.
     )
)
;Set dialog's initial size
WinRESIZE 20. 5.
STOP
DIALOG.END
ENDDO
```

A



B



A Click "Options>>" to expand the dialog

B Dialog shown expanded. Click "<<Options" to shrink the dialog.

Control Positioning

Control placement starts at (0, 0) which is the top left of the dialog. The default size is 9 units wide and 1 unit high. After a control is placed the next position is calculated by advancing the Y co-ordinate by one and keeping the same X co-ordinate. The size defaults to the same size as the last placed control. To override the size and position of the next control to be placed, use the **POS** command.

POS [<x>] [<y>] [*<width>*] [*<height>*]	Defines the size and position of the next control to be placed on a dialog window.

<x>	0 - 16383.5 units. Can be specified in 0.5 unit increments.
<y>	0 - 8191.75 units. Can be specified in 0.25 unit increments.
<width>	0 - 16383.5 units. Can be specified in 0.5 unit increments.
<height>	0 - 8191.75 units. Can be specified in 0.25 unit increments.

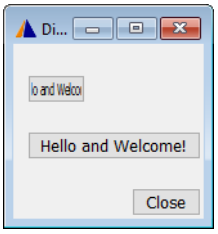
,	The value of the previous POS argument is used. In the example here the previous value for the `<width>` is preserved.
	`;  <x>  <y>  <width>  <height>`
	`POS  3.  7.  ,  2.`

<no_argument>	The value of the previous POS argument is used, starting from right to left. In the example below, the values for `<width>` and `<height>` will be taken from the previous POS command.
	`;  <x>  <y>  <width>  <height>`
	`POS  3.  7.`

The examples above have a period "." after the numbers in the arguments to **POS**. This instructs TRACE32 to treat these as decimal numbers.

Controls with text in them will attempt to fit the text to the size of the control.

```
DIALOG.view
(
   POS 1. 1. 5. 1.
   BUTTON "Hello and Welcome!" ""
   POS 1. 3. 15. 1.
   BUTTON "Hello and Welcome!" ""
   POS 10. 5. 6. 1.
   BUTTON "Close" "CONTinue"
)
STOP
DIALOG.END
ENDDO
```



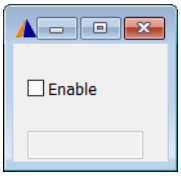
Some dialog controls must be associated with a PRACTICE label. This provides some extra capabilities:

- PRACTICE script can enable or disable the control.
- The PRACTICE script can extract user input or control state information via the label.
- The PRACTICE script can set a value into the control by using the label.
- The PRACTICE script can set the control's state.
- If a control has a command block associated with it, this can be executed.

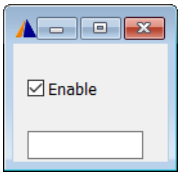
DIALOG.Enable *<label>*	Enable the control which is associated with *<label>*.
DIALOG.Disable *<label>*	Disable the control which is associated with *<label>*.

```
DIALOG
(
     POS 1. 1. 10. 1.
CB1: CHECKBOX "Enable"
     (
        IF DIALOG.BOOLEAN(CB1)
            DIALOG.Enable T1
        ELSE
            DIALOG.Disable T1
     )
     POS 1. 2. 10. 1.
T1:  EDIT "" ""
)
;Set starting state for controls after the DIALOG command but before
;the STOP.
DIALOG.DISABLE T1
STOP

ENDDO
```

A



B



A When the dialog opens the **CHECKBOX** is enabled and the **EDIT** control is not.

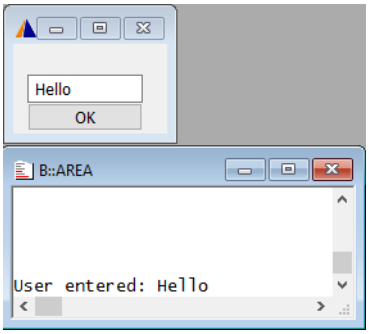
B Clicking the **CHECKBOX** will enable the **EDIT** control.

DIALOG.BOOLEAN(<label>)	Returns the boolean value of the dialog control associated with *<label>*.

Information from a control is made available to the PRACTICE script through the label associated with that control. The method of extraction is to use the **DIALOG.STRing()** function.

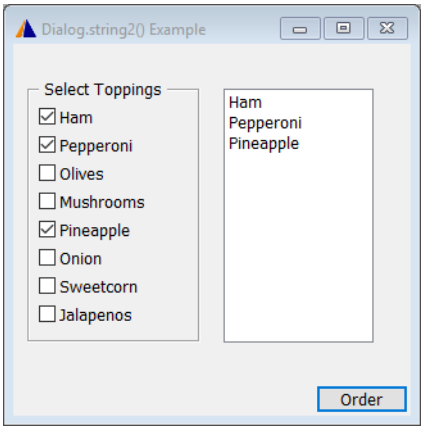
DIALOG.STRing(<label>)	Returns the data from the dialog control associated with *<label>*.
DIALOG.STRing2(<label>)	Retrieves the complete list of values from a list dialog control: **LISTBOX**, **MLISTBOX**, **DLISTBOX**, **COMBOBOX**, etc. Returned values are in comma-separated string.

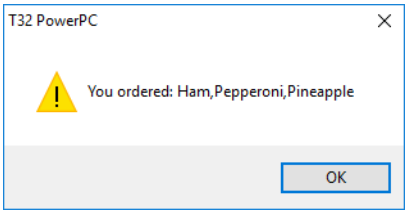
```
LOCAL &userInput

DIALOG.view
(
     POS 1. 1. 10. 1.
TXT: EDIT "" ""
     POS 1. 2. 10. 1.
     BUTTON "OK" "CONTinue"
)
STOP
&userInput=DIALOG.STRing(TXT)
AREA.view
PRINT "User entered: &userInput"
ENDDO
```



```
LOCAL &toppings &ordered
DIALOG.view
(&
     HEADER "Dialog.string2() Example"
     POS 1. 1. 15. 10.
     BOX "Select Toppings"
     POS 2. 2. 13. 1.
CB1: CHECKBOX "Ham"
     (
        GOSUB check_item "Ham"
     )
CB2: CHECKBOX "Pepperoni"
     (
        GOSUB check_item "Pepperoni"
     )
CB3: CHECKBOX "Olives"
     (
        GOSUB check_item "Olives"
     )
CB4: CHECKBOX "Mushrooms"
     (
        GOSUB check_item "Mushrooms"
     )
CB5: CHECKBOX "Pineapple"
     (
        GOSUB check_item "Pineapple"
     )
CB6: CHECKBOX "Onion"
     (
        GOSUB check_item "Onion"
     )
CB7: CHECKBOX "Sweetcorn"
     (
        GOSUB check_item "Sweetcorn"
     )
CB8: CHECKBOX "Jalapenos"
     (
        GOSUB check_item "Jalapenos"
     )
     POS 18. 1.5 13. 9.
TOP: MLISTBOX "" ""
     POS 26. 12. 8. 1.
     DEFBUTTON "Order" "CONTinue"
)
STOP
```

```
&ordered=DIALOG.STRing2(TOP)
DIALOG.OK "You ordered: &ordered"
DIALOG.END
ENDDO

check_item:
  ENTRY &newitem
  &newitem=&newitem       ;Remove any extra quotes
  IF "&toppings"==""      ;Is this the first item to be added to the list
     &toppings="&newitem"
  ELSE
     &toppings="&toppings"+","+"&newitem"
  DIALOG.Set TOP "" "&toppings"

RETURN
```



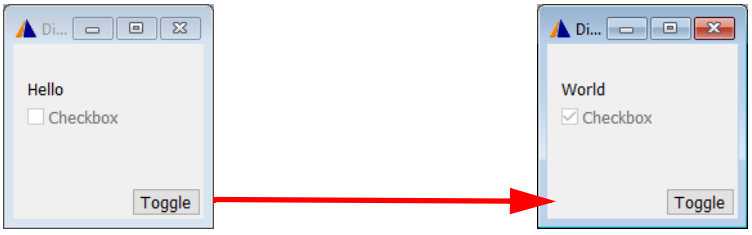


Setting a value or state to a control

Controls that can be associated with a label can be set or have data entered into them using the **DIALOG.Set** command.

DIALOG.Set *<label> <value>*	Set the *<value>* to the control associated with *<label>*.

```
DIALOG.view
(
    POS 1. 1. 5. 1.
T1: DYNTEXT "Hello"
    POS 1. 2. 15. 1.
C1: CHECKBOX "Checkbox" ""
    POS 10. 5. 6. 1.
    BUTTON "Toggle"
    (
      IF DIALOG.BOOLEAN(C1)
      (
         DIALOG.Set T1 "Hello"
         DIALOG.Set C1 FALSE()
      )
      ELSE
      (
         DIALOG.Set T1 "World"
         DIALOG.Set C1 TRUE()
      )
    )
)
DIALOG.Disable C1
STOP
DIALOG.END
ENDDO
```

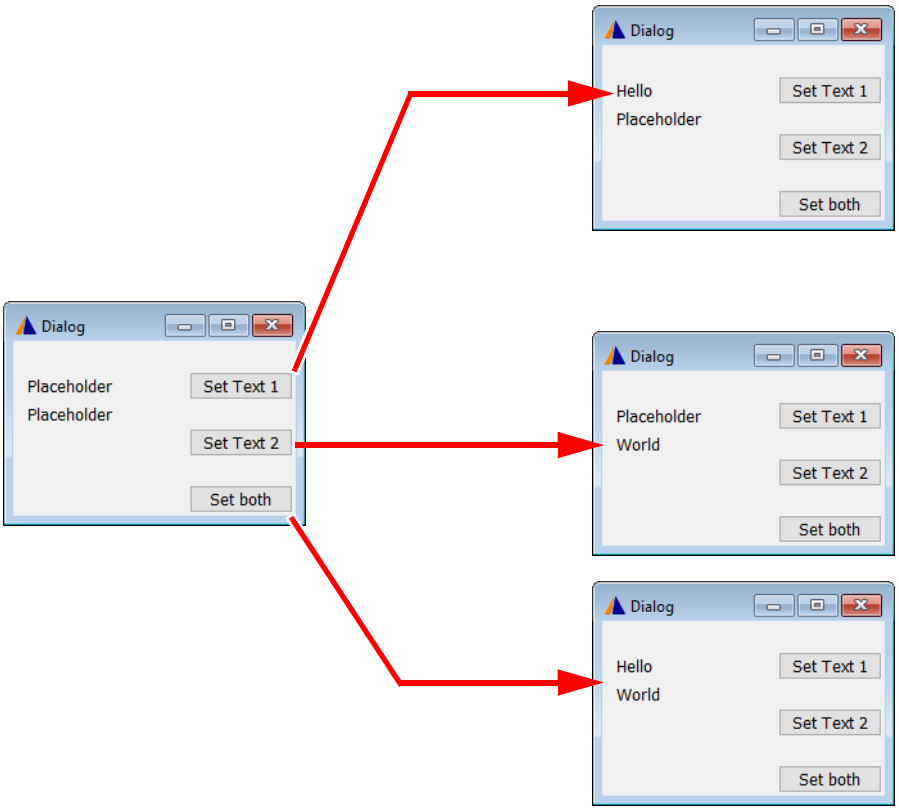


Execute a command

It is possible to execute the command or block associated with any other control. This is done using the **DIALOG.EXecute** command.

DIALOG.EXecute *<label>*	Execute the command of the control which is associated with `<label>`.

```
DIALOG.view
(
    POS 1. 1. 10. 1.
T1: DYNTEXT "Placeholder"
    POS 1. 2. 10. 1.
T2: DYNTEXT "Placeholder"
    POS 15. 1. 9. 1.
B1: BUTTON "Set Text 1"
    (
       DIALOG.Set T1 "Hello"
    )
    POS 15. 3. 9. 1.
B2: BUTTON "Set Text 2"
    (
      DIALOG.Set T2 "World"
    )
    POS 15. 5. 9. 1.
B3: BUTTON "Set both"
    (
      DIALOG.EXecute B1
      DIALOG.EXecute B2
    )
)

STOP
DIALOG.END
ENDDO
```


Dialog
Placeholder
Placeholder
Set Text 1
Set Text 2
Set both
Dialog
Hello
Placeholder
Set Text 1
Set Text 2
Set both
Dialog
Placeholder
World
Set Text 1
Set Text 2
Set both
Dialog
Hello
World
Set Text 1
Set Text 2
Set both

File Browsing

PRACTICE provides several commands for file and directory browsing. There are two broad categories: those that return a value like a subroutine and those that can set the returned value to a label.

DIALOG.DIR *<directory_name>*	Creates a dialog box to choose a directory name. *<directory_name>* must contain wildcard characters.
DIALOG.File *<file>*	Creates a file chooser dialog box. *<file>* must contain wildcard characters. The file must exist and the user must have read access to the file. The file name returned contains the full path.
DIALOG.File.SAVE *<file>*	Creates a file chooser dialog box. *<file>* must contain wildcard characters. The file need not exist but the user must have permissions to create the file in the selected directory. The file name returned contains the full path.

```
LOCAL &filename
DIALOG.File.open *.cmm
ENTRY %LINE &filename              ;Use ENTRY to obtain result from
                                   ;DIALOG.FILE. Added %LINE option in
                                   ;case the file name had spaces in it.
PRINT "&filename selected."

;To extract just the file name
PRINT "File name is "+OS.FILE.NAME(&filename)

;To extract the path information
PRINT "Path details are "+OS.FILE.PATH(&filename)
ENDDO
```

The above example also uses the **OS.FILE.NAME**() and **OS.FILE.PATH**() functions to extract the file name and file path respectively.

DIALOG.SetDIR *<label> <dir_path>*	Creates a dialog box to choose a directory name. *<dir_path>* must contain wildcard characters. The result is placed into the *<label>*.
DIALOG.SetFile *<label> <file_name>*	Creates a file chooser dialog box. *<file_name>* must contain wildcard characters. The file must exist and the user must have read access to the file. The result will be placed into the **EDIT** control associated with *<label>*.
DIALOG.SetFile.SAVE *<label> <file_name>*	Creates a file chooser dialog box. *<file_name>* must contain wildcard characters. The file need not exist but the user must have permissions to create the file in the selected directory. The result will be placed into the **EDIT** control associated with *<label>*.

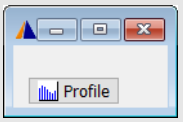
```
LOCAL &fname
DIALOG.view
(
      POS 1. 1. 25. 1.
FILE: EDIT "" ""
      POS 30. 1. 5. 1.
B:    BUTTON "..." "DIALOG.SetFile FILE *"
      POS 30. 3. 5. 1
      BUTTON "OK" "CONTinue"
)
STOP
&fname=DIALOG.STRing(FILE)
DIALOG.END
PRINT "File - &fname"
ENDDO
```

The **DIALOG.SetFile** command has been attached to the BUTTON. Commands can be attached to some controls. If the command is a single line it can be enclosed in quotes ("). If the command is more complex it can be a PRACTICE block (enclosed in parentheses).

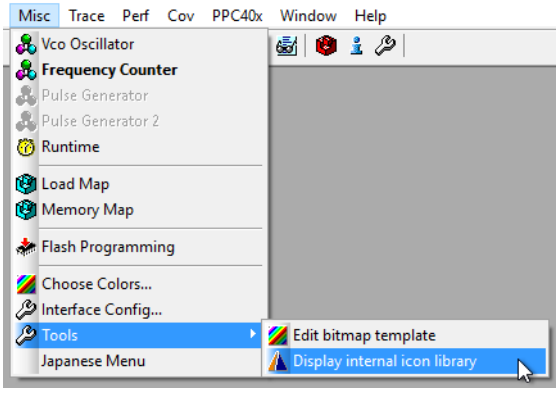
Icons

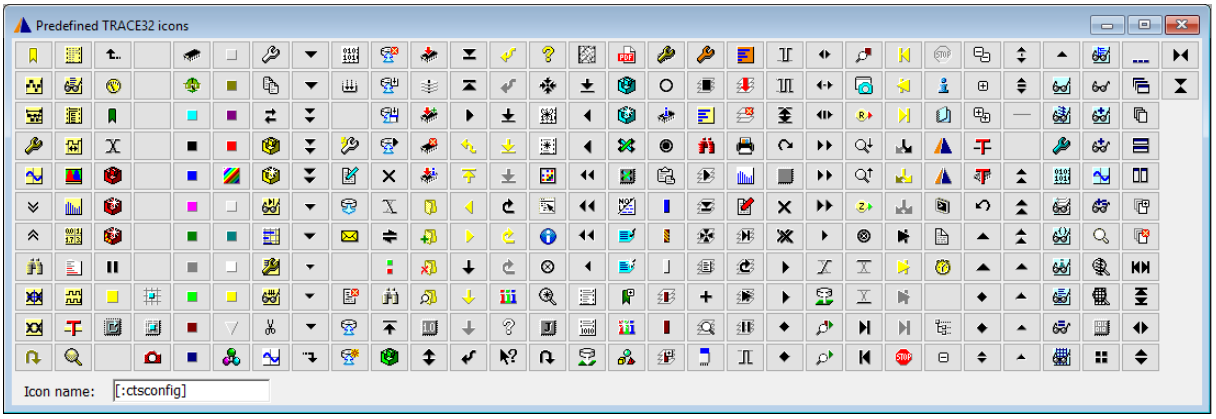
TRACE32 has a number of built-in icons which can be used in dialog or menu programming. These are added by using the syntax **[:**<icon_name>**]** in the text part of a control.

```
DIALOG.view
(
    POS 1. 1. 8. 1.
    BUTTON "[:profile]Profile" ""
)
STOP
ENDDO
```



A list of internal icons can be displayed. Select the icon and the name is displayed.





Dialog Example

A complete list of dialog controls can be found in **"PowerView Command Reference"** (ide_ref.pdf). In this section we will examine a few of them in the context of a complex example. The full version of this script can be found in ~~/demo/practice/dialogs/dialog_example_generic.cmm

First, declare some local macros to be used in the script. Some of them are given default values. Some lines have been split using the "\" character to make the script more readable here. Some assignments, like `&instr` are made using multiple assignments to build up a longer value.

```
LOCAL &bond &batman &captain &bondsel &captainsel &batmansel &expand \
&progress &instr &myname

&bond="Sean Connery,George Lazenby,Roger Moore,Timothy Dalton,"
&bond="&bond"+"Pierce Brosnan,Daniel Craig"
&batman="Adam West,Michael Keaton,Val Kilmer,George Clooney,"
&batman="&batman"+"Christian Bale,Ben Affleck"
&captain="Jonathon Archer,James Kirk,Jean-Luc Picard,"
&captain="&captain"+"Benjamin Sisko,Katherine Janeway"

&expand=0.
&progress=0.

&instr="Instructions for USE"+CONVert.CHAR(10.)
&instr="&instr"+"============="+CONVert.CHAR(10.)
&instr="&instr"+"1) Enter your name. We will not send you spam"
&instr="&instr"+" e-mail ;-)"+CONVert.CHAR(10.)
&instr="&instr"+"2) Select an option in the ""Options"" pane." \
+CONVert.CHAR(10.)
&instr="&instr"+"3) Double-click to select an item from the list." \
+CONVert.CHAR(10.)
&instr="&instr"+"4) Once an item from each category has been"
&instr="&instr"+" selected, click ""OK""."+CONVert.CHAR(10.)
```

Display the dialog. The characters "`&+`" after the opening parenthesis allow for runtime macro substitution into the dialog controls. The code below will create a header and add a text line and edit field to the dialog.

```
DIALOG.view
(&+
        HEADER "Dialog Example"
        POS 1. 0. 20. 1.
        TEXT "Please enter your name:"
        POS 22. 0. 24. 1.
MYNAME: DEFEDIT "" ""
```

The next few lines add a box and a group of radio buttons (**CHOOSEBOXes**) into the box pane. Each **CHOOSEBOX** is part of the same group. The label name takes the form *<group>*.*<member>* and TRACE32 ensures that only one member of each group can be selected at any one time. Each **CHOOSEBOX** has a command associated with it which will be executed when the user selects that control.

The command has the characters "&-" after the opening parenthesis to disable macro substitution during the execution of the command. This prevents a double substitution taking place on the macros used in the command. Each command block checks to see if a selection for that group has already been made. If it hasn't the list is displayed. If a selection has already been made the list of items is displayed and the selected one highlighted. This is done with the two forms of the **DIALOG.Set** command.

```
          POS 1. 1. 20. 5.
          BOX "Options"
          POS 2. 2. 15. 1.
O.BOND:   CHOOSEBOX "James Bond"
          (&-
            IF "&bondsel"==""
              DIALOG.Set SEL "" "&bond"
            ELSE
              DIALOG.Set SEL "&bondsel" "&bond"
          )
O.BMAN:   CHOOSEBOX "Batman"
          (&-
            IF "&batmansel"==""
              DIALOG.Set SEL "" "&batman"
            ELSE
              DIALOG.Set SEL "&batmansel" "&batman"
          )
O.CAPN:   CHOOSEBOX "Star Trek"
          (&-
            IF "&captainsel"==""
              DIALOG.Set SEL "" "&captain"
            ELSE
              DIALOG.Set SEL "&captainsel" "&captain"
          )
```

Here is the code that draws the list and initially populates it with the &bond list. When an item in the list is double-clicked by the user the command is executed and, again, macro substitution is temporarily disabled. The current member of the **CHOOSEBOX** group is determined and the macro for the selection from that group is updated. Then one of the **CHECKBOXes** is checked to indicate that a selection for that group has been made. A macro containing the percentage completion is updated and this is used to update the progress bar. Once progress exceed 99% the **OK** button on the dialog is enabled, using **DIALOG.Enable**.

```
        POS 22. 1.5 25. 4.
SEL:    LISTBOX "&bond"
        (&-
           IF DIALOG.BOOLEAN(O.BOND)
           (
              &bondsel=DIALOG.STRing(SEL)
              DIALOG.Set B_SEL "ON"
           )
           ELSE IF DIALOG.BOOLEAN(O.BMAN)
           (
              &batmansel=DIALOG.STRing(SEL)
              DIALOG.Set BM_SEL "ON"
           )
           ELSE
           (
              &captainsel=DIALOG.STRing(SEL)
              DIALOG.Set C_SEL "ON"
           )
           &progress=&progress+34.
           DIALOG.Set PBAR &progress
           IF &progress>99.
              DIALOG.Enable BOK
        )
```

This excerpt draws a box with three **CHECKBOXes** and a progress bar in it. The **CHECKBOX** controls will be disabled and only updated via the script to prevent un-wanted user interaction.

```
        POS 1. 6. 45. 3.
        BOX "Selection made"
        POS 2. 7. 12. 1.
B_SEL:  CHECKBOX "Bond Selected" ""
        POS 18. 7. 12. 1.
BM_SEL: CHECKBOX "Batman Selected" ""
        POS 32. 7. 12. 1.
C_SEL:  CHECKBOX "Captain Selected" ""
        POS 1. 9. 45. 3.
        BOX "Progress"
        POS 2. 10. 43. 1.
PBAR:   BAR
```

Here, a **TREEBUTTON** is used to expand or reduce the size of the dialog. This allows for a set of hidden instructions to be made visible.

```
          POS 1. 12. 1. 1.
TB1:      TREEBUTTON ""
          (&-
             IF &expand==0
             (
               WinRESIZE 48. 18.
               &expand=1
               DIALOG.Set TB1 "ON"
             )
             ELSE
             (
               WinRESIZE 48. 13.
               &expand=0
               DIALOG.Set TB1 "OFF"
             )

          )
```

This code adds a **TEXTBUTTON** next to the **TREEBUTTON** providing the user with a larger area to click. The **TEXTBUTTON** uses **DIALOG.EXecute** to execute the command of the **TREEBUTTON**.

```
          POS 2. 12. 10. 1.
          TEXTBUTTON "Instructions"
          (
             DIALOG.EXecute TB1
          )
```

This code draws a button with the text “OK” and which will execute the command **CONTinue** when clicked by the user. It then creates an **INFOTEXT** control which will display the text stored in macro `&instr`. The other options to **INFOTEXT** configure the border style, font and background color. The closing parenthesis indicates the end of the dialog controls.

```
          POS 40. 12. 8. 1.
BOK:      BUTTON "OK" "CONTinue"
          POS 1. 13.5 46. 4.
INFO:     INFOTEXT "&instr" STICKER SUNKEN Variable1 7.
)
```

After the end of the dialog controls, there are some statements that set the initial state of some of the controls. The **CHECKBOX** controls are disabled. The initial member of the group of **CHOOSEBOXes** is set. The dialog is resized to hide the **INFOTEXT** control and the final **OK** button is disabled.

```
DIALOG.Set O.BOND
DIALOG.Disable B_SEL
DIALOG.Disable BM_SEL
DIALOG.Disable C_SEL
WinRESIZE 48. 13.
DIALOG.Disable BOK
STOP
```

The user's name is extracted from the **EDIT** control and a message box is displayed showing their choices. When the user closes the message box (**DIALOG.OK**), the dialog itself will be closed and the script will terminate.

```
&myname=DIALOG.STRing(MYNAME)
DIALOG.OK "Thanks &myname. Your selections were:" "" "Favorite James Bond
is &bondsel" \
"Favorite Batman is &batmansel" "Favorite Captain is &captainsel"
DIALOG.END
ENDDO
```

PRACTICE in a Multi-Core Environment

PRACTICE scripts need some modification to function effectively in a modern multicore debug environment. This section contains advice and guidance on how to achieve this. Where the target is configured for Symmetric Multi-Processing (SMP), a single instance of TRACE32 PowerView is used to control all cores. In this case many PRACTICE commands add an option **/CORE** *<n>* option to indicate that the command should be run with reference to a particular core or cores. This section will focus now on Asymmetric Multi-Processing (AMP) systems where multiple instances of TRACE PowerView are used to control heterogenous cores.

For simple multicore debug sessions it may be sufficient to have one instance of TRACE32 PowerView configure the main core and then launch a second instance which would attach to and configure the other debug session. Such a system might look like the examples below.

```
;Script for core 1
SYStem.CPU XXX    ;Set correct CPU type
SYStem.Up
Data.LOAD.Elf myprog.elf

; Let core 1 run until it has initialised core 2
Go.direct core_2_release
WAIT !STATE.RUN()

; Launch a second instance of TRACE32 with a custom config file and
; start-up script
OS.screen t32mxxx -c config_core2.t32 -s start_up_script-core2.cmm

ENDDO
```

```
; script for core 2
SYStem.CPU xxx
SYStem.Mode Attach
Break.direct
WAIT !STATE.RUN()
Data.LOAD.Elf * "myprog.elf"
ENDDO
```

Some systems require a more flexible approach, especially where communication between instances of PowerView is required.

Communication via InterCom

TRACE32 PowerView implements a communication system called INTERCOM which allows instances to communicate with each other via UDP. More information about this can be found at **"InterCom"** in PowerView Command Reference, page 181 (ide_ref.pdf). This requires a small modification to the configuration file (normally `~~/config.t32`) file to enable. A modified file is shown below with the changes highlighted.

```
PBI=
USB
CORE=1

IC=NETASSIST
PORT=20001
```

The `PORT=` indicates that this instance of TRACE32 PowerView will listen for incoming requests on this UDP port. The ports must be unique.

A separate configuration file is needed for each instance of TRACE32 with a unique `CORE=` and `PORT=` identifier. Alternatively, a single generic configuration file may be used and the values passed in via the command line, for example:

```
IC=NETASSIST
PORT=${1}

PBI=
${2}
CORE=${3}
${4}
```

Start with:

```
T32<exe> -c <config file> <port> <connection> <core> <node name>
t32mppc  -c config.t32     20001      USB        1    NODE=t32ppc1
t32mtpu  -c config.t32     20002      USB        2    NODE=t32tpu1
```

Below is an example of launching other instances from a PRACTICE script.

```
PRIVATE &t32path &t32mppc &t32mtpu
;Get TRACE32 executable directory
&t32path=OS.PresentExecutableDirectory()

; Create macros for executables
&t32mppc="&t32path/t32mppc"
&t32mtpu="&t32path/t32mtpu"

; Launch other instances
OS.screen "&t32mppc" -c ~~/config.t32 20002 USB 2 NODE=2nd_PPC_core
OS.screen "&t32mtpu" -c ~~/config.t32 20003 USB 3 NODE=eTPU_core

ENDDO
```

The main script can wait until the other instances are available by using the **InterCom.WAIT** command. The example above would be extended like this.

```
; Extension of previous script
OS.screen "&t32mppc" -c ~~/config.t32 20002 USB 2 NODE=2nd_PPC_core
OS.screen "&t32mtpu" -c ~~/config.t32 20003 USB 3 NODE=eTPU_core

INTERCOM.WAIT localhost:20002
INTERCOM.WAIT localhost:20003

ENDDO
```

Now that all instances are synchronised, commands can be issued to each instance. This can be done using the **InterCom.execute** command and can be wrapped in a PRACTICE macro to make the script more readable.

```
; Using INTERCOM.execute
SYStem.CPU e200z6
INTERCOM.execute localhost:20002 SYStem.CPU e200z6
INTERCOM.execute localhost:20003 SYStem.CPU e200z2

; Using a macro
LOCAL &core2 &core3
&core2="INTERCOM.execute localhost:20002"
&core3="INTERCOM.execute localhost:20003"
SYStem.CPU e200z6
&core2 SYStem.CPU e200z6
&core3 SYStem.CPU e200z2
```

Designing Robust PRACTICE Scripts

A script may run on the PC of the person who developed it but it may not run as well (or at all) on the PC of a colleague or a customer. This section will help you to design PRACTICE scripts that will be more robust and more portable.

There are several things to take into consideration when designing a robust PRACTICE script.

- Host PCs may have different directory structures.
- Host PCs may be running different Operating Systems.
- Different debug hardware may be used.
- The script may be run with a different target board or processor.
- Differences in TRACE32 versions.
- Differences in TRACE32 settings.

Scripts should avoid hard coding any paths; users will likely have their own PCs set out to their own tastes. Network drives may also be mapped to different letters.

This example will only work if the ELF file is located in the current working directory.

```
; This will only work if the ELF file is located in the current directory
Data.LOAD.Elf "myprog.elf"
```

This example will work only if the ELF file is located in this path. It will not work on a colleague's PC who has stored the project in `C:\users\barry\Project_1\`.

```
; This will fail if the ELF file is located somewhere else.
; But it will leave the current working directory unchanged.
; ELF file path "C:\users\rico\project_1\out\myprog.elf"

PRIVATE &pwd
&PWD=OS.PresentWorkingDirectory()
ChDir "C:\users\rico\Project_1\out"
Data.LOAD.Elf "myprog.elf"
ChDir "&pwd"
```

OS.PresentWorkingDirectory()	Returns the name of the working directory as a string.

An alternative approach would use relative directory paths. TRACE32 provides a number of functions or built-in path prefixes. For example:

```
; 1st example uses OS.PresentPracticeDirectory() to get the current
; script's directory
PRIVATE &ppd
&ppd=OS.PresentPracticeDirectory()
Data.LOAD.Elf "&ppd/myprog.elf"
```

```
; 2nd approach uses built in shortcut "~~~~" instead.
Data.LOAD.Elf "~~~~/myprog.elf"
```

OS.PresentPracticeDirectory()	Returns the name of the directory where the current PRACTICE script is located as a string. `~~~~/` is the TRACE32 path prefix equivalent.

If the script is located in the same directory as the ELF file, the above examples will work. If the ELF file is in a sub-directory, this can also be made to work.

```
; 1st example uses OS.PresentPracticeDirectory() to get the current
; script's directory
PRIVATE &ppd
&ppd=OS.PresentPracticeDirectory()
Data.LOAD.Elf "&ppd/out/myprog.elf"
```

```
; 2nd approach uses built in shortcut "~~~~" instead.
Data.LOAD.Elf "~~~~/out/myprog.elf"
```

Users could always be presented with a file chooser to browse for the ELF file to be loaded.

```
Data.LOAD.Elf "*.elf"
```

The script could be called with the top level directory for the project as an argument.

```
; Call script as:
; DO load_script.cmm C:\user\rico\project
;
PARAMETERS &basedir
Data.LOAD.Elf "&basedir/out/myprog.elf"
```

A better version would be to check if the file exists and prompt the user to browse for it if it is not where expected.

```
PRIVATE &ppd &file
&ppd=OS.PresentPracticeDirectory()
&file="&ppd"+"/myprog.elf"
IF OS.FILE("&file")==TRUE()
    Data.LOAD.Elf "&file"
ELSE
(
    &file="&ppd"+"/out/myprog.elf"
    IF OS.FILE("&file")==TRUE()
       Data.LOAD.Elf "&file"
    ELSE
       Data.LOAD.Elf "*.elf"
)
```

OS.FILE(<filename>)	Returns TRUE if the file exists.

A complete list of all path prefixes is provided in **"Path Prefixes"** in PowerView User's Guide, page 51 (ide_user.pdf).

Host Operating System

There will be differences between host Operating Systems. Scripts should be written in such a way as to remove the impact of these differences from the user.

If a script uses a forward slash ("/") in a path name, TRACE32 will automatically use the correct slash for the underlying host Operating System.

```
Data.LOAD.Elf "~~~~\myprog.elf"     ; Not Portable - Windows only!

Data.LOAD.Elf "~~~~/myprog.elf"     ; Portable - Host OS Independent
```

Use the TRACE32 built-in commands for file system manipulation. These will automatically be resolved to the correct commands for the host Operating System. A full list of these commands an be found in See **"File and Folder Operations"** in PowerView User's Guide, page 77 (ide_user.pdf) but include:

REN *<filename>*	Rename file.
COPY *<source> <destination>*	Copy file.
MKDIR *<path>*	Create directory.
RMDIR *<path>*	Delete directory.
ZIP *<source>* [*<destination>*]	Compress files.

The built-in macro "~~" which resolves to the TRACE32 system directory can be used to overcome differences in host Operating Systems.

```
; Will only work on Windows host with default installation
COPY C:\t32\config.t32 C:\t32\config-usb.t32

;Will only work on Linux or MacOS with default installation
COPY /opt/t32/config.t32 /opt/t32/config-usb.t32

;Will work anywhere regardless of installation directories
COPY "~~/config.t32" "~~/config-usb.t32"
```

Different Operating Systems use different line termination sequences.

- Windows uses CR + LF
- Linux and MacOS use LF

TRACE32 uses LF to denote an end of line when reading a file. The following code will read a file on any host11

```
LOCAL &text
OPEN #1 "~~~~/log.txt"
READ #2 %LINE &text
WHILE !EOF()
(
    PRINT "&text"
    READ #2 %LINE &text
)
CLOSE #2
```

When TRACE32 writes a file, it uses the correct end of line sequence for the current host Operating System. Scripts can be made to write portable files by using this trick.

```
OPEN #2 "~~~~/log.txt" /Create
WRITE #2 %String "Hello" %Ascii 0x0A
WRITE #2 %String "World" %Ascii 0x0A
CLOSE #2
```

The function **OS.VERSION(0)** can be used to determine the host operating system. Using this allows scripts to adapt to underlying fundamental differences. An example can be found on page RICO.

Debug Hardware

A script may need to adapt depending upon the debug hardware that is used. Some scripts may not be appropriate for certain hardware. The script should check and then inform the user before quitting. A number of functions exist for TRACE32 hardware detection and more information can be found in **"General Function Reference"** (general_func.pdf). A few are listed below. Each returns TRUE if the relevant hardware is detected.

- **AUTOFOCUS()**
- **hardware.COMBIPROBE()**
- **hardware.POWERDEBUG()**
- **hardware.POWERTRACE2()**
- **INTERFACE.SIM()**

```
IF INTERFACE.SIM()
(
    DIALOG.OK "Simulator mode not supported by this script." \
          "Press 'OK' to exit the script."
    ENDDO
)

IF !AUTOFOCUS()
(
    DIALOG.OK "Autofocus pre-processor required." \
          "Please connect correct hardware." \
          "Press 'OK' to exit the script."
    ENDDO
)
;Rest of the script goes here
```

Target CPU and Board

The function **CPUFAMILY()** will return the name of the family. Any core which can be debugged with the same t32m* executable is part of the family. Each family will have its own set of unique functions.

```
IF CPUFAMILY()=="ARM"
(
    IF CABLE.TWOWIRE()
        SYStem.CONFIG SWD
)
```

The function **CPU()** will return the value of the processor selected by the user in the **SYStem.state** window.

The JTAG ID code of the *n*th device in the scan chain can be obtained by using the **IDCODE(<n>)** function.

```
LOCAL &device
SYStem.RESet
SYStem.CPU ARM7TDMI                // Default CPU
SYStem.DETECT IDCode
&device=IDCODE(0)&0x0fffffff

IF &device==0x0B95C02F             // TI OMAP4430 (PandaBoard)
(
   SYStem.CPU OMAP4430APP1
   SYStem.CONFIG CTIBASE 0xd4148000
   SYStem.Mode Up
)
ELSE IF &device==0x049220DD        // Altera Excalibur
(
   SYStem.CPU EPXA
   SYStem.Mode Up
   Data.Set C15:00000001 %LE %Long 0x0178 // disable Instruction Cache
)
ELSE
(
   PRINT %ERROR "Don't know device 0x" %Hex &device
   ENDDO
)
```

Some CPUs do not support byte addressable memory; the smallest addressable unit may be 16, 24 or 32 bits. The script below provides a means of determining this value.

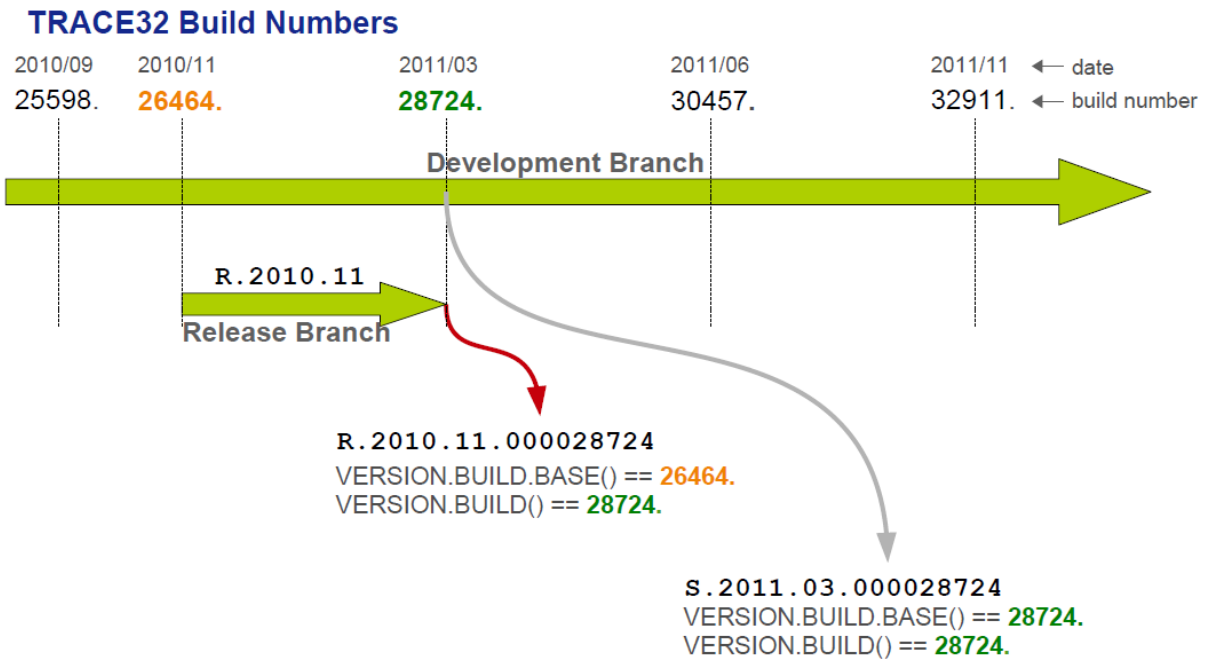
```
LOCAL &width

Data.Set VM:0 %Byte 0 1 2 3 4 5 6 7 8
&width=Data.Byte(VM:1)
PRINT "Width of one address is " %Decimal &width ". byte(s)"
```

TRACE32 Version

A script should never assume which TRACE32 PowerView features are available to it; it always a good idea to check. The two main functions for this are:

VERSION.BUILD()	Returns the build number of TRACE32. Features up to this build may be included in this version.
VERSION.BUILD.BASE()	Returns the base build number for this instance of TRACE32 PowerView. All features of this build will be included.

The diagram here explains the relationship between the two different build numbers.




```
IF VERSION.BUILD()<56572.
(
    DIALOG.OK "This script requires TRACE32 build greater than 56572."
    ENDDO
)
```

TRACE32 Settings

TRACE32 is highly customizable and a script should make no assumptions about how a user has their software configured.

Scripts should not assume a default radix as this is a CONTinue option and users will set it according to their own preference.

```
Data.Set D:10 %Byte 42

; Will write 0x42 to address 0x10 if the radix is set to hex
; Will write 0x2A to address 0x0A if the radix is set to decimal
```

A script should always enforce the radix that it wishes to use to avoid such confusion.

```
Data.Set D:0x10 %Byte 0x42     ; Force hex values
Data.Set D:10. %Byte 42.       ; Force decimal values
```

It is strongly recommended that scripts format any PRACTICE macros before printing them to make the values un-ambiguous to users. Some examples are shown below.

```
LOCAL &var &text
&var=0x42+23.
&text="The result is 0x"+FORMAT.HEX(0,&var)
PRINT "&text"
```

```
LOCAL &var
&var=0x42+23.
PRINT "The result is 0x"+FORMAT.HEX(0,&var)
```

```
LOCAL &var
&var=0x42+23.
PRINT "The result is 0x" %Hex &var
```

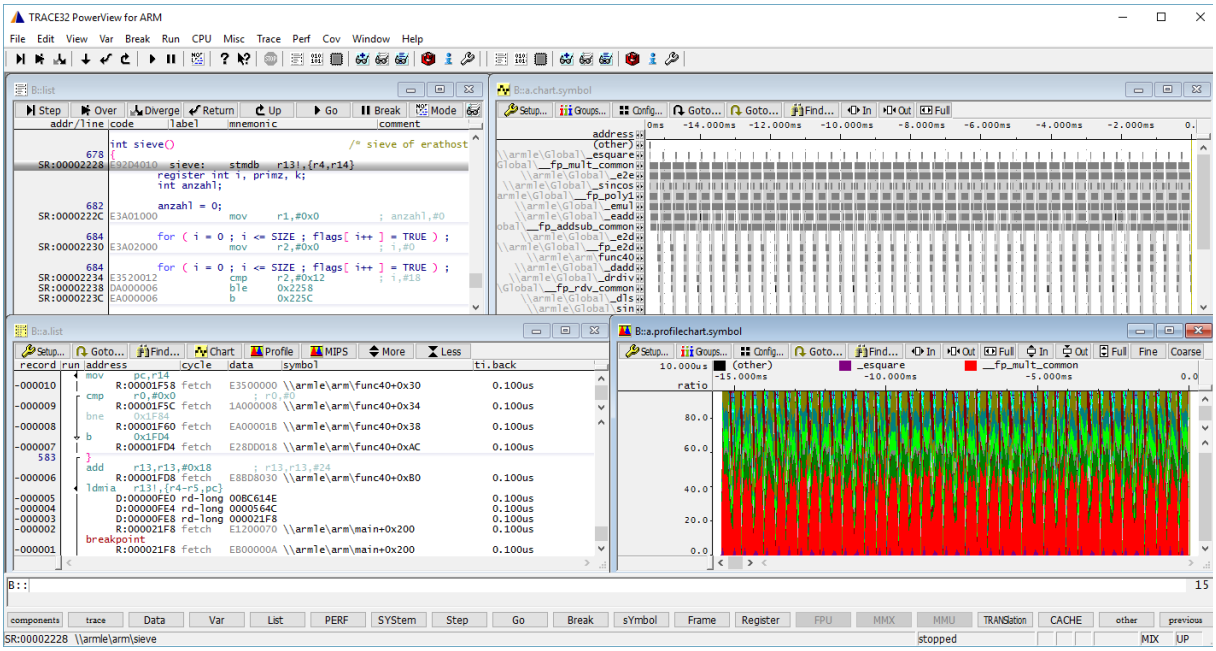
Three formatting functions exist: **FORMAT.HEX()**, **FORMAT.Decimal()** and **FORMAT.BIN()**.

If a script intends to make use of breakpoints or will open more than a single window, it may be a good idea to store the users' current settings and then restore them after the script has completed its work This can be done using the **STOre** command.

```
PRIVATE &bpfile &winfile

; Store user's Breakpoints and Windows away so we can retrieve them
; later. Get temporary files from the host OS
&bpfile=OS.TMPFILE()
&winfile=OS.TMPFILE()

; We'll need to manually add the ".CMM" extension
&bpfile="&bpfile"+".cmm"
&winfile="&winfile"+".cmm"

; Store Breakpoints to temporary file
STOre "&bpfile" Break.direct

; Now clear all the existing Breakpoints
Break.Delete /ALL

; Store Windowss to temporary file
STOre "&winfile" Windows

; Now clear all the existing Windows
WinCLEAR


; Run the rest of the script here


; Restore the user's breakpoints and windows
DO "&bpfile"
DO "&winfile"
ENDDO
```

If the script needs to open any additional windows to display some results it would be unwise to assume that the script user has a display of a certain size or resolution. Absolute window positioning and sizing may not produce a readable display. The WinPOS command can take percentages as well as absolute values as its arguments. Windows could be positioned and sized as a percentage of the current size of the TRACE32 window.

```
WinCLEAR

WinPOS 0% 0% 40% 50%
List.auto

WinPOS 40% 0% 60% 50%
Analyzer.Chart.sYmbol

WinPOS 0% 50% 50% 50%
Analyzer.List

WinPOS 50% 50% 50% 50%
Analyzer.PROfileChart.sYmbol

ENDDO
```



An alternative approach would be to determine the local screen resolution and open a pre-configured set of windows which match the current screen size. An example of how to do this has been provided in `~~/demo/practice/screen_size.cmm` which attempts to extract the screen resolution from the

host OS. An example use case would look like this.

```
PRIVATE &scrw &scrh

; Call the script which will return the values for screen width and
; screen height
DO ~~/demo/practice/screen_size.cmm
RETURNVALUES &scrw &scrh

; the returned values can be printed like this
PRINT "Screen Width = &scrw"
PRINT "Screen Height = &scrh"

; This handy little trick for PRACTICE will convert the 'string' macro
; into a 'numerical' macro so it can be used in a test
&scrw="&(scrw)"

IF &scrw<1280.
(
  PRINT "Loading Small screen layout."
  ; Here, either call a script with windows optimized for a
  ; small screen or GOSUB to a routine which will open the correct window
  ; layout
)
ELSE
(
  PRINT "Loading Large screen layout."
  ; Call correct script or GOSUB to routine which opens windows for a
  ; higher resolution screen.
)

ENDDO
```

If a PRACTICE script encounters an error it will halt at the line that produced the error. Using the command **SETUP.WARNSTOP** it is possible to also configure a PRACTICE script to halt on a command that merely produces a warning. More advanced error handling can be used in scripts so that they can recover more gracefully should something fail. This relies on the event driven features of PRACTICE and in it's simplest form looks like the example here. The script will keep running even though an error will be produced.

```
; Enable error handler
ON ERROR CONTinue

; Execute a command that will fail
Data.LOAD.Elf "non_existent_file.elf"

; Restore previous error handler
ON ERROR inherit
ENDDO
```

The call to `ON ERROR inherit` will restore the previous error handler; error handlers may be stacked. A better error handler might look like this.

```
; Enable error handler
ON ERROR GOSUB
(
  ; Tell the user that the file wasn't loaded and ask them to browse
  ; for its correct location.
  PRINT "File not found."
  Data.LOAD.Elf *
  RETURN
)

; Execute a command that will fail
Data.LOAD.Elf "non_existent_file.elf"

; Restore previous error handler
ON ERROR inherit
ENDDO
```

Since build 72159 it is possible to check after each command whether or not an error occurred. This allows PRACTICE script developers to work in a more traditional style. To prevent the script from halting, this should be combined with the error handler described previously.

```
IF VERSION.BUILD()<72159.
(
  DIALOG.OK "Your version of TRACE32 is too old to run this script."
  ENDDO
)

ON ERROR CONTinue

; label to return to for re-try
start_here:

; Clear any existing errors
ERROR.RESet
Data.LOAD.Elf "myprog.elf"
; Check if there was an error
IF ERROR.OCCURRED()
(
  ; Try to determine what the error was and fix it
  IF ERROR.ID()=="#FILE_ERRNFD"
  (
    PRINT "File not found"
    Data.LOAD.Elf *
  )
  IF ERROR.ID()=="#emu_errpwrf"
  (
    DIALOG.OK "Please apply power to the target then click 'OK'"
    GOTO start_here
  )
  ; Finally, an unexpected error. Report it to the user.
  IF ERROR.ID()!=""
  (
    DIALOG.OK "Error occurred : " ERROR.ID()
  )
)

ENDDO
```

Designing a script to be re-useable requires that some thought is given to checking argument values passed in by the user. The script cannot assume that the arguments it gets will always be correct. Using **STRing.SCANAndExtract()** can ensure that argument order does not matter but does not enforce the contents of an argument. PRACTICE macros can always be treated as strings and something like the `IsNum` sub-routine below can check that a macro only contains a numerical value in either decimal or hex format.

```
; Example sub routine for checking a macro is numeric only
LOCAL &arg1 &arg2 &ret
&arg1=25.
&arg2="Hello"

GOSUB IsNum "&arg1"
RETURNVALUES &ret
PRINT "&ret"                ; Prints TRUE()

GOSUB IsNum "&arg2"
RETURNVALUES &ret
PRINT "&ret"                ; Prints FALSE()

ENDDO

IsNum:
  PARAMETERS &txt
  LOCAL &ret &len &i &char &hex
  &ret=TRUE()

  &len=STRing.LENgth("&txt")

  IF STRing.ComPare(STRing.LoWeR("&txt"),"0x*")==TRUE()
  ( ; Possibly a hex number
    &i=2.
    &hex=TRUE()
  )
  ELSE
  ( ; Assume a decimal number
    &i=0.
    &hex=TRUE()
  )

  RePeaT (&len-&i)
  (
    &char=STRing.MID(STRing.LoWeR("&txt"),&i,1)
    IF &hex==TRUE()&&STRing.FIND("0123456789.abcdef","&char")==FALSE()
      &ret=FALSE()
    ELSE IF STRing.FIND("0123456789.","&char")==FALSE()
      &ret=FALSE()
    &i=&i+1
  )
RETURN "&ret"
```

Creating a Custom Command

It is possible to create a custom command in TRACE32. This is done by using the command **GLOBALON** .

```
GLOBALON CoMmanD <name> DO <script>
```

The command *<name>* is created but is restricted to a maximum of 9 characters. In the example script below, call it without arguments to register the command, call it with a single argument of "REMOVE" to remove the command.

```
PARAMETERS &test &testfile &count

IF "&test"==""
(
  ; No arguments so register the command
  ; Command name is limited to 9 characters
  LOCAL &this_script
  &this_script=OS.PresentPracticeFile()
  GLOBALON CoMmanD TESTRUN DO "&this_script"
)
ELSE IF "&test"=="REMOVE"
(
  ; Use this to remove the global command
  GLOBALON CoMmanD TESTRUN
)
ELSE
(
  RePeaT &count
  (
    GOSUB &test "&testfile"
  )
)
ENDDO

test1:
  PARAMETERS &tf
  PRINT "Running test1 with data file (&tf)"
RETURN

test2:
  PARAMETERS &tf
  PRINT "Running test2 with data file (&tf)"
RETURN
```

Once the command has been registered it can be accessed form the TRACE32 command line or from within another script, for example.

```
; Call the script with no arguments to register the command
DO custom_command.cmm

; Use the command to run the test cases
TESTRUN "test1" "testdata1.txt" "0x10"
TESTRUN "test2" "testdata2.txt" "0x03"

; Call the script again to remove the command
DO custom_command.cmm "REMOVE"
ENDDO
```

Common Pitfalls

This final section focuses on some of the common mistakes that can be made in PRACTICE scripts.

Try not to use blocking commands unless you are debugging your script. Commands like **STOP** and **ENTER** will stop the script from executing but will give the user no indication that the script has halted or is awaiting further input. It is a better idea to use a dialog instead.

Watch for white space. It is required after an **IF** or **WHILE** command but should not appear in expressions.

```
; INCORRECT example
IF(&i<5.)

; CORRECT examples
IF (&i<5.)
IF &i<5.
```

```
; INCORRECT expression
&x = ( 5. + 8. * 2 ) / ( 3 + &i )

; CORRECT expression
&x=(5.+8.*2.)/(3/+&i)
```

Functions should not appear in filenames.

```
; INCORRECT example
Data.LOAD.Elf STRing.TRIM("&dir")+"/myprog.elf"

; CORRECT example
&dir=STRing.TRIM("&dir")
Data.LOAD.Elf "&dir/myprog.elf"
```

Always use the correct declaration of PRACTICE macros. Review the difference between **LOCAL** and **PRIVATE**.

Always assume that the default radix is not what you require. Force your macros to the correct radix. For example, use 0x10 or 16. but never 10 on it's own.

Use the built-in commands for TRACE32 directories instead of hard coding paths to scripts or files.

Use TRACE32 built-in commands for host OS file manipulation (copy, change directory, delte, etc.). This makes the script portable across different host Operating Systems.