LAUTERBACH
DEVELOPMENT TOOLS

PRACTICE Script Language Reference Guide

Release 02.2024

MANUAL

PRACTICE Script Language Reference Guide

PRACTICE Script Language Reference Guide

Version 04-Mar-2024

History

02-Nov-22	In the chapter ‘Related Documents‘ a reference to ide_user.pdf has been added.

Related Documents

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

For information about how to pass parameters, PRACTICE macros, etc., refer to:

- **“PRACTICE Script Language User’s Guide”** (practice_user.pdf)

For information about literals, operands, operators, and operator precedence, refer to:

- **“PowerView User’s Guide”** (ide_user.pdf). Alternatively, choose **Help** menu > **Index**, and then enter the search item.

For information about functions, refer to:

- **“PowerView Function Reference”** (ide_func.pdf)
- **“General Function Reference”** (general_func.pdf)

For information about the purpose of functions in TRACE32, how to use functions, and the difference between functions and commands, refer to:

- **“General Function Reference”** (general_func.pdf)

```
;To retrieve the same information via the TRACE32 command line:
HELP.Index "scripting"
HELP.Index "literals"
HELP.Index "parameter types"
HELP.Index "operands"
HELP.Index "operators"
HELP.Index "operator precedence"
HELP.Index "functions"
```

APPEND — Append to file

Format: **APPEND** *<file> <parameter_list>*

Appends data to a file. The file is created if it does not exist.

The syntax of the command is similar to the **PRINT** command.

Example:

```
APPEND datafile.dat "Test"
```

See also

- CLOSE
- MKTEMP
- OPEN
- WRITE

▲ 'Release Information' in 'Legacy Release History'

BEEP — Acoustic signal

Format: **BEEP**

Generates an acoustic signal on the host computer.

See also

- SETUP.SOUND

CLOSE — Close file

Format:	**CLOSE #**<*buffer_number*>

Closes an input or output file.

Example:

```
OPEN  #1 ~~~/test.dat /Write ; open file for writing
WRITE #1 "Test data"         ; write data to file
CLOSE #1                     ; close file

TYPE ~~~/test.dat            ; optional: open file in TYPE window
```

The path prefix ~~~ expands to the temporary directory of TRACE32.

See also

- APPEND
- OPEN
- READ
- WRITE
- WRITEB
- Data.WRITESTRING
- Var.WRITE

CONTinue — Continue PRACTICE script

Format:	**CONTinue** [[<*line*>] [<*file*>]]

A PRACTICE script which has been stopped will be restarted at the current PRACTICE command. PRACTICE scripts will be stopped by the **STOP** command or by a breakpoint within the script.

The **CONTinue** command can also be used to resume a script that has been halted due to an error condition. The wrong command may be replaced by an interactive command.

<*line*>	Line number. Go till <*line*> in the active PRACTICE script (*.cmm).
, <*file*>	Line number is omitted. Path and file name of PRACTICE script. Go till first executable line in the PRACTICE script <*file*>.
<*line*> <*file*>	Go till <*line*> in the PRACTICE script <*file*>.

Example 1:

```
DO test.cmm                    ; start script
…
…
…                              ; script stopped at breakpoint
CONTinue                       ; continue
```

Example 2:

```
…
STOP                           ; stop script by STOP command
…
…
CONTinue                       ; continue
```

See also

- END
- PBREAK
- STOP

DECRYPT — Decrypts a text or binary file

Format:	**DECRYPT** *<keystring>* *<encrypted_file>* [*<decrypted_file>*]

Uses the original key string to decrypt a text or binary file previously encrypted with the **ENCRYPT** command. The resulting file can get a new name or replace the old file.

See also

- ENCRYPT
- ▲ 'Encrypt/Execute Encrypted Files' in 'PowerView User's Guide'

Format:	**DO** *<file>* [*<parameter_list>*]

Starts a PRACTICE script (*.cmm). The **DO** command can be used on the command level to start a PRACTICE script or within a script to run another file like a subroutine. PRACTICE files started by a **DO** command should be terminated by the **ENDDO** command. Additional parameters may be defined which are passed to the subroutine. The subroutine reads the parameter list using the **ENTRY** command.

Using the **DO** command even those settings saved by the **STOre** command can be retrieved.

<file>	The default extension for *<file>* is ***.cmm**. The default extension can be changed with the command **SETUP.EXTension PRACTICE**.

Examples:

```
; store window setting to the PRACTICE file 'test.cmm'
STOre test.cmm Win

; set up window setting by executing the script 'test.cmm'
DO test.cmm
```

```
; Endless loop with subroutine call
&count=1
WHILE TRUE()
(
     DO mem_test
     PRINT "MEMTEST " &count
     &count=&count+1
)
ENDDO
```

```
ChDir.DO c:\sample\x.cmm        ; change to c:\sample and execute the
                                ; file x.cmm
```

See also

- DODECRYPT
- ENTRY
- ON
- STOre
- ChDir
- GLOBALON
- PSTEP
- END
- GOSUB
- RePeaT
- ENDDO
- JUMPTO
- RUN

▲ 'Release Information' in 'Legacy Release History'
▲ 'Introduction to Script Language PRACTICE' in 'Training Script Language PRACTICE'

Format:	**DODECRYPT** *<keystring>* *<encrypted_file>* [*<parameter_list>*]

Executes a PRACTICE script (*.cmm) that was encrypted with same key string by the command **ENCRYPTDO**. The key string is necessary for execution. The PRACTICE script file stays encrypted.

See also

- DO

▲ 'Encrypt/Execute Encrypted Files' in 'PowerView User's Guide'

ECHO Write text and data to an AREA window (with format decoration)

[Examples]

Format:	**ECHO** [{**%**<*attribute*>}] [{{**%**<*format*>} <*data*>}]
<*attribute*>:	**AREA** <*area_name*> **CONTinue** **HOME**
<*format*>:	<*type*> **COLOR.**<*color*> **ERROR** **WARNING**
<*type*>:	**Ascii** | **BINary** | **Decimal** | **Hex** | **String**
<*color*>:	**NORMAL** | **BLACK** | **MAROON** | **GREEN** | **OLIVE** | **NAVY** | **PURPLE** **TEAL** | **SILVER** | **GREY** | **RED** | **LIME** | **YELLOW** | **BLUE** | **FUCHSIA** **AQUA** | **WHITE**

Writes the given arguments to the default **AREA A000** or the selected **AREA** window. When writing to the default **AREA A000**, the written data is also shown in the TRACE32 message line.

What is the difference between the commands ...?

ECHO	PRINT
Writes all data decorated to indicate the format of the data.	Writes all data without any format decoration (e.g. without the prefix “0x” for hexadecimal numbers).
For a comparison of the different outputs, see examples.	

<*attribute*>, <*format*>, etc.	For descriptions of the command arguments, see **PRINT**.

Examples

The following table shows the output in a message area for the same data written with **ECHO** and **PRINT**:

<data>	AREA output with ECHO	AREA output with PRINT
0x042	`0x42`	`42`
%Hex 66.	`0x42`	`42`
23.	`23.`	`23`
%Decimal 0x17	`23.`	`23`
0y110011	`0y00110011`	`110011`
%BINary 0x33	`0y00110011`	`00110011`
'X'	`'X'`	`X`
%Ascii 0x58	`'X'`	`X`
5==5	`TRUE()`	`TRUE`
5==3	`FALSE()`	`FALSE`
"text"	`text`	`text`
P:0x001000	`P:0x1000`	`P:0x1000`
500ms	0.5000000000s	0.5000000000s
DATE.MakeUnixTime(1990.,10.,3,0,0,0)	654912000.	654912000
Var.VALUE(23 * 47)	0x439	439

See also

- PRINT
- PRINTF
- SPRINTF

▲ 'Release Information' in 'Legacy Release History'

Format:	**ELSE** [**IF** *<condition>*]

The command or script block following an **ELSE** statement will be executed, if the condition in the previous **IF** statement is false.

Examples:

```
IF Register(PC)==ADDRESS.OFFSET(main)
     PRINT "PC on start position"
ELSE
     PRINT "PC not on start position"
…
```

```
IF Register(PC)==ADDRESS.OFFSET(main)
(
     PRINT "PC on start position"
     STOP
)
ELSE IF Register(PC)==ADDRESS.OFFSET(end)
(
     PRINT "PC on end position"
     STOP
)
ELSE
(
     PRINT "PC neither on start nor on end position"
     Register.Set PC main
     STOP
)
```

See also

- IF

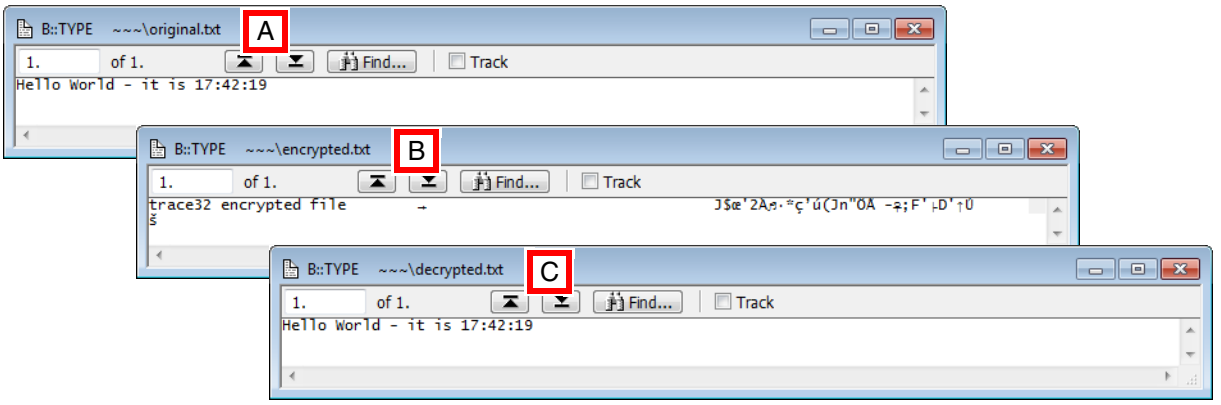
Format:	**ENCRYPT** *<keystring>* *<source_file>* [*<encrypted_file>*]

Encrypts the contents of a text or binary file using the specified key string. If no file name for the encrypted file is specified, the original file will be replaced by the encrypted file. The resulting file can be decrypted with the command **DECRYPT**, together with the original key string.

NOTE: Do **not** encrypt PRACTICE scripts (*.cmm) or PER (*.per) files with **ENCRYPT**.
- For encrypting PRACTICE scripts use **ENCRYPTDO**.
- For encrypting PER files use **ENCRYPTPER**.

Example:

```
;let's write to a newly-created file and display the file [A]
OPEN  #1 ~~~\original.txt /CREATE
WRITE #1 "Hello World - it is " DATE.TIME()
CLOSE #1
TYPE     ~~~\original.txt

;let's encrypt and display the file [B]
ENCRYPT  "123456789" ~~~\original.txt  ~~~\encrypted.txt
TYPE     ~~~\encrypted.txt

;let's now decrypt and display the file [C]
DECRYPT  "123456789" ~~~\encrypted.txt ~~~\decrypted.txt
TYPE     ~~~\decrypted.txt
```



See also

■ ENCRYPTDO ■ ENCRYPTPER ■ DECRYPT

▲ 'Encrypt/Execute Encrypted Files' in 'PowerView User's Guide'

Format:	**ENCRYPTDO** *<keystring>* *<source_file>* [*<encrypted_file>*]

Encrypts a PRACTICE script using the specified key string. If no file name for the encrypted file is specified, the original file will be replaced by the encrypted file.

The encrypted PRACTICE script can be executed with the command **DODECRYPT** using the original key string. Decrypting the PRACTICE script or viewing its original file contents in plain text is not possible.

Use **ENCRYPTDO** to generate PRACTICE scripts which can be executed by the end user, without the possibility to read or modify the script.

NOTE:	Do not use **ENCRYPTDO** on already encrypted scripts!

Example:

```
;encrypt a PRACTICE script file in the system directory of TRACE32
ENCRYPTDO "987654321"   ~~~/secret.cmm    ~~~/secret_encrypted.cmm

;execute the encrypted PRACTICE script file
DODECRYPT "987654321"   ~~~/secret_encrypted.cmm
```

See also

- ENCRYPT
- ENCRYPTPER

▲ 'Encrypt/Execute Encrypted Files' in 'PowerView User's Guide'
▲ 'Release Information' in 'Legacy Release History'

Format:	**ENCRYPTPER** *<keystring>* *<source_file>* [*<encrypted_file>*]

Encrypts a PER definition file using the specified key string. If no file name for the encrypted file is specified, the original **PER** file will be replaced by the encrypted PER file.

The encrypted PER file can be *executed* and viewed with the command **PER.viewDECRYPT** using the original key string. Decrypting the PER file or viewing its original file contents in plain text is not possible.

Use **ENCRYPTPER** to generate PER files which can be executed by the end user, without the possibility to read or modify the original PER file contents.

NOTE:	Do not use **ENCRYPTPER** on already encrypted PER files!

Example:

```
;encrypt a PER file residing in the system directory of TRACE32
ENCRYPTPER "123456789" ~~\pera940t.per ~~\pera940t_encrypted.per

;execute the encrypted PER file, expand the section "ID Registers"
PER.viewDECRYPT "123456789" ~~\pera940t_encrypted.per "ID Registers"
```

See also

- ENCRYPT
- ENCRYPTDO
- PER.viewDECRYPT

▲ 'Encrypt/Execute Encrypted Files' in 'PowerView User's Guide'

Format:	**END**

Executes the actions listed below and is typically used in a PRACTICE script file (*.cmm). Interactive usage at the TRACE32 command line is needed to clear the PRACTICE stack after a stack overrun has occurred.

Actions executed by the **END** command:

- Terminates all PRACTICE scripts.
- Affects the PRACTICE stack, which can be viewed with **PMACRO.list**, as follows:
  - The local PRACTICE stack is cleared, e.g. command extensions, error exits, **LOCAL** and **PRIVATE** PRACTICE macros.
  - The global PRACTICE stack is retained. That is, **GLOBAL** PRACTICE macros and **GLOBALON** events remain on the stack, unless **PMACRO.RESet** is executed.
- Closes all custom dialogs.
- Closes all files opened with the **OPEN** command.

Example:

```
END
```

See also

- ENDDO
- CONTinue
- DIALOG
- DO
- PMACRO.list
- PMACRO.RESet
- STOP
- GLOBALON

Format:	**ENDDO** [*<return_value_list>*]

Ends a PRACTICE script. Execution is continued in the calling script. If no calling script file exists, the PRACTICE script execution will be stopped.

It is recommended to end all PRACTICE script files (*.cmm) with **ENDDO** to remove them from the PRACTICE stack.

Examples:

```
; Sub-module memory test
Data.Test 0x0--0x0fff
Data.Test 0x8000--0x0ffff
ENDDO
```

PRACTICE scripts can pass return values to the caller:

```
; script test_status.cmm

ENDDO TRUE()                                    ; return TRUE as result
; ENDDO FALSE()                                 ; return FALSE as result
```

```
; script enddo_param.cmm

DO test_status                                  ;execute test_status.cmm
ENTRY &result                                   ;read result

IF &result                                      ;react on result
     DIALOG.OK "Test passed"
ELSE
     DIALOG.OK "Test failed"

ENDDO
```

NOTE:	**TRUE()** and **FALSE()** are PRACTICE functions returning the corresponding boolean values.

See also

- END
- DO
- RUN
- STOP

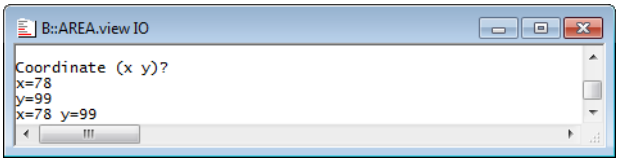
Format:	**ENTER** *<parlist1>* | *<parlist2>* | *<parlist3>*
<parlist1>:	[**%LINE**] *<macro>*
<parlist2>:	{*<macro>*}
<parlist3>:	{*<macro>*} **%LINE** *<macro>*

Lets you pass arguments via a special PRACTICE I/O window to PRACTICE macros (see the **AREA** command group). Arguments are separated by blanks.

<parlist1>	With the **%LINE** option, the entire line is read into the PRACTICE macro.
<parlist2>	The number of arguments passed via the PRACTICE I/O window must match the number of PRACTICE macros. Otherwise, an error occurs.
<parlist3>	With the **%LINE** option, surplus arguments are assigned to the last PRACTICE macro as one line.

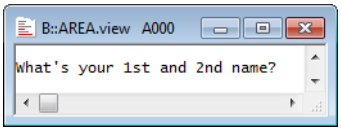
Example 1: <parlist1> - without %LINE

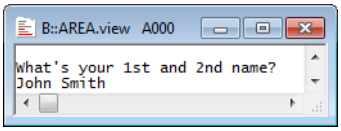
```
AREA.Create IO                              ;create an AREA window
AREA.Select IO  ;select the AREA window as PRACTICE input/output window

AREA.view IO                                ;display the AREA window
PRINT "Coordinate (x y)? "                  ;print input request

PRINT "x="
ENTER &x                                    ;wait for first user input
PRINT "y="
ENTER &y                                    ;wait for second user input

PRINT "x=" &x " y=" &y                      ;print x value and y value

SCREEN.WAIT 1.s                             ;wait 1 second, and then
WinCLEAR TOP                                ;close the AREA window
AREA.RESet                                  ;reset the AREA window system
```

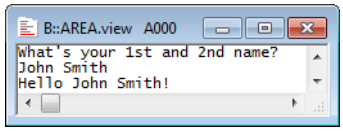


Example 2: <parlist1> - with %LINE

```
LOCAL &name                                 ;declare PRACTICE macro
AREA.Select A000                            ;let's use the default AREA
                                            ;A000 in this example
AREA.view A000                              ;open AREA window

PRINT "What's your 1st and 2nd name? "      ;print input request
PRINT ""                                    ;print an empty line
ENTER %LINE &name                           ;wait for user input

PRINT "Hello &name!"
```

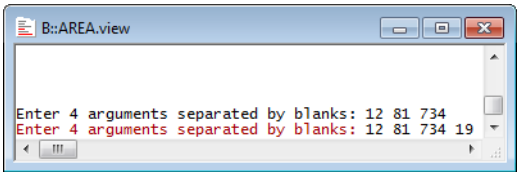


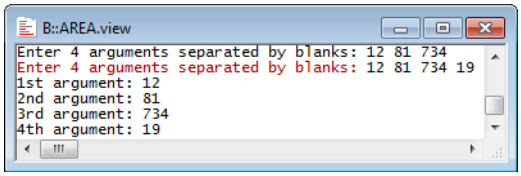




Example 3: <parlist2> - with Multiple PRACTICE Macros

```
LOCAL &a &b &c &d                           ;declare 4 PRACTICE macros
AREA.view                                   ;open AREA window

PRINT "Enter 4 arguments separated by blanks: "
myLabel: ENTER &a &b &c &d                  ;prompt user for input
                                            ;all 4 macros are initialized?
IF ("&a"==("")||("&b"==("")||("&c"==("")||("&d"==("")
(
   PRINT %ERROR "Enter 4 arguments separated by blanks: "
   GOTO myLabel                             ;prompt user again
)
ELSE
(                                           ;print to the AREA window
   PRINT "1st argument: &a"
   PRINT "2nd argument: &b"
   PRINT "3rd argument: &c"
   PRINT "4th argument: &d"
)
```





See also

- AREA
- INKEY

▲ 'I/O Commands' in 'Training Script Language PRACTICE'

Format:	**ENTRY** *<parlist>*
<parlist>:	[**%LINE**] *<macroname>*

Passing of parameters to or from PRACTICE scripts/subroutines. Arguments are separated by blanks. With the **%LINE** option the entire line is read into one PRACTICE macro.

Without a **LOCAL** command for defining local PRACTICE macros, existing PRACTICE macros from preceding routines are used. Only not existing PRACTICE macros are defined automatically.

Example:

```
; TRACE32 PowerView command line

DO test.cmm P:0x1000
```

```
; contents of PRACTICE script test.cmm

SYStem.Up
;...                              ; Take argument, here P:0x1000, from
ENTRY &address                    ; the PRACTICE script call

GOSUB func1 &address 1.           ; Call subroutine func1 with two args

ENTRY &result                     ; Get return value of subroutine
PRINT "Result=" &result

ENDDO

func1:
     LOCAL &addr &size            ; Define local PRACTICE macros
     ENTRY &addr &size            ; Get arguments from subroutine call

     &size=&size-1.
     Data.Set &addr++&size 0x2    ; Execute command
     &retval=Data.Byte(&addr)     ; Calculate return value

     RETURN &retval               ; Return value to caller
```

See also

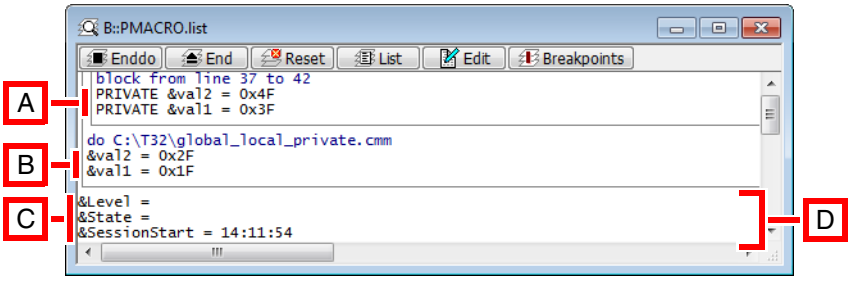
- DO
- GLOBALON
- GOSUB
- LOCAL
- PARAMETERS
- PRIVATE
- RETURN
- RETURNVALUES
- PRACTICE.ARG()

▲ 'Release Information' in 'Legacy Release History'
▲ 'Parameter Passing' in 'Training Script Language PRACTICE'

GLOBAL — Create global PRACTICE macro

Format: **GLOBAL** {*<macro>*}

Creates a global macro. Global macros are visible everywhere. They are not erased when the declaring file or block ends. That is, global macros remain in the global PRACTICE stack after execution of the procedure or subroutine.



A Private macros

B Local macros

C Global macros

D Global PRACTICE stack frame

Example: This script shows how to declare and initialize global PRACTICE macros. Stepping through the code allows you to see how global macros behave in relation to local and private macros (See "**How to...**").

```
PMACRO.list                                ;View the PRACTICE stack

GLOBAL &SessionStart &State &Level         ;Declare three global macros
LOCAL &val1 &val2                          ;Declare two local macros

&SessionStart=CLOCK.TIME()                 ;Initialize a global macro
&val1=0x1f                                 ;Initialize the local macros
&val2=0x2f

(                                          ;Open a sub-block
   PRIVATE &val1 &val2                     ;Declare private macros

   &val1=0x3f                              ;Initialize private macros
   &val2=0x4f
)                                          ;Close sub-block
```

See also

- LOCAL
- PMACRO.EXPLICIT
- PMACRO.IMPLICIT
- PMACRO.list
- PMACRO.RESet
- PRIVATE
- SPRINTF

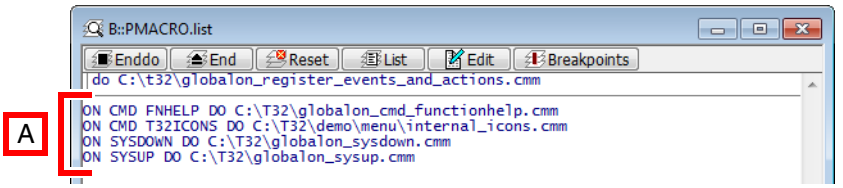
Format:	**GLOBALON** *<event>* [*<action>*]
<event>:	*<device_specific_events>* *<practice_specific_events>* *<cpu_specific_events>*
<practice_specific_events>:	**ALWAYS** **ERROR** **STOP** **CMD** *<command_name> <action>* **TIME** *<delay> <action>*
<action>:	**DO** *<file>* **EXECute** *<trace32_command>* (only available for **CMD**)

The **GLOBALON** command enables the automatic start or branching of the PRACTICE programs controlled by several events. In order for events and their actions to be available, they need to be registered in TRACE32. To register events and their actions, you can for example:

- Run the **GLOBALON** commands via the TRACE32 command line.
- Include the **GLOBALON** commands in the PRACTICE script file system-settings.cmm. As a result, they are automatically registered when you start TRACE32. For more information, see **"Automatic Start-up Scripts"** (practice_user.pdf).
- Include the **GLOBALON** commands in any other script. As a result, they are only registered when you run that script.

Registered actions remain stored on the global PRACTICE stack frame. Therefore, the actions are valid for the entire duration of the TRACE32 session, or until they are removed manually.

The currently active actions can be viewed with the **PMACRO** command. The outermost frame is the global PRACTICE stack frame, as shown below.



A Global PRACTICE stack frame with **GLOBALON** commands

Events: <practice_specific_events> for the GLOBALON Command

PRACTICE specific Events	Descriptions
ALWAYS	The defined PRACTICE sequence is executed permanently, as long as no keyboard input occurs or no normal PRACTICE script is activated.
ERROR	Will be executed if a syntax or runtime error occurs in PRACTICE. The default action of this event is to halt script execution.
STOP	Will be executed, when the STOP button from the toolbar is pushed. Warning: A PRACTICE script using this feature may hang and cannot be stopped then by the STOP button!
CMD *<command_name>* *<action>*	Definition of a user-defined command for TRACE32. The length of the *<command_name>* is limited to 9 characters and the character set [0..9], [@..Z], [a..z], '_', '+' and '-'. Arguments can be passed to the user-defined command. • For available *<actions>*, see below. • For examples, see below.
TIME *<delay>* *<action>*	Executes the script after a delay time. For a description of the *<action>* **DO** *<file>*, see below.

Events: <device_specific_events>

<device_specific_ events>	For a description of device-specific events, refer to the **GLOBALON** command (general_ref_g.pdf).

Events: <cpu_specific_events>

<cpu_specific_ events>	Debuggers providing CPU-specific events are listed in the **See also** block at the end of the **GLOBALON** command description (general_ref_g.pdf).

<actions> for the GLOBALON Command

One of the following actions can be defined for any of the above events:

Actions	Descriptions
no action specified	An already defined action for a particular global event will be removed from the global PRACTICE stack frame. See “Unregistering GLOBALON Commands”.
DO *<file>*	If the event occurs, the specified PRACTICE script *<file>* will be executed automatically.
EXECute *<trace32_command>*	If the event occurs, the specified single-line *<trace32_command>* will be executed automatically. Unlike the action **DO** *<file>*, the action **EXECute** *<trace32_command>* is only intended for frequently-used and complex single-line commands; see example below. The individual *<actions>* do *not* require their own dedicated PRACTICE script files (*.cmm). Instead, you can maintain as many **GLOBALON… EXECute…** commands as you want in just one PRACTICE script file. Additionally, you can create and modify the **GLOBALON… EXECute…** commands via the TRACE32 command line. **NOTE**: This *<action>* is only available for the *<practice_specific_event>* **CMD**.

Examples

The following examples show how you can use the TRACE32 command **GLOBALON** to create, register and unregister your own user-defined commands.

Example 1: TICONS

This is a very simple example for demo purposes. It creates the user-defined command `TICONS`, which opens the TRACE32 icon library.

Register your user-defined command `TICONS`, e.g. by copying and pasting the following **GLOBALON CMD** into the TRACE32 command line:

```
; Register the user-defined command TICONS
;              <command>      <action>
GLOBALON CMD   TICONS   DO "~~/demo/menu/internal_icons.cmm"
```

Result: Typing `TICONS` at the TRACE32 command line now opens the TRACE32 icon library.

NOTE:	Built-in commands cannot be overwritten while the debugger is active, i.e. a user-defined command named **SYS** is not possible. **SYS** will continue to open the **SYStem.state** window.

Example 2: GLOBALON CMD without an Argument

The user-defined command `SOURCE` displays the HLL source code without changing your emulation mode setting ASM, HLL, or MIX; see **DEBUGMODE()** and **Mode**.

1. Develop the action, i.e. a PRACTICE script (*.cmm), such as the following one:

```
;Check if the window named myWin01 already exists to prevent
;duplicate windows
IF !WIN.EXIST(myWin01)
(
     ;Additionally, assign a user-defined window position and name
     WinPOS 0% 0% 100% 50% , , , myWin01

     ;Display the program in source format (HLL, high level
     ;language)
     List.HLL /Track
)
ENDDO
```

2. Register your user-defined command and its action in TRACE32.

```
GLOBALON CMD SOURCE DO "~~~/globalon_cmd_source.cmm"
```

Example 3: GLOBALON CMD with an Argument

The user-defined command `SOURCE2` accepts an argument if you enter one at the TRACE32 command line and passes it to the *<action>*.

1. Develop the action, i.e. a PRACTICE script (*.cmm), such as the following one:

```
; Displays the program in source format (HLL, high-level language)
; Starts the listing at the symbol passed as an argument,
; e.g. at main

LOCAL &myArg
ENTRY &myArg       ; Get the argument the user has entered
                   ; at the TRACE32 command line

; Check if an argument is passed or not
IF "&myArg"==""
    List.HLL /Track
ELSE
    List.HLL &myArg /Track

ENDDO
```

2. Register your user-defined command and its action in TRACE32.

```
GLOBALON CMD SOURCE2 DO "~~~/globalon_cmd_source2.cmm"
```

Example 4: GLOBALON CMD with the EXECute <command>

The following script registers the two user-defined commands **`TL`** and **`LMPC`** on the global PRACTICE stack frame. The TRACE32 command that is actually executed when the user types **`TL`** or **`LMPC`** at the command line is formatted in blue. Note that the backslash `\` is the line continuation character.

To try, simply copy and paste the script into the TRACE32 command line. Then type just **`TL`** or **`LMPC`** to open a **Trace.List** or **List.Mix** window as specified in the blue command string.

```
GLOBALON CMD TL EXECute WinExt.WinResist.WinLarge.Trace.List \
%TimeFixed TIme.Zero DEFault /Track

GLOBALON CMD LMPC EXECute List.Mix Register(PC) /Track /MarkPC
```

Unregistering GLOBALON Commands

You can unregister all **GLOBALON** commands or just a selected **GLOBALON** command.

NOTE:	Unregistering all **GLOBALON** commands from the global PRACTICE stack frame also deletes all global PRACTICE macros.

- To unregister all **GLOBALON** commands, type at the TRACE32 command line:

```
END           ; Ends all active PRACTICE scripts
PMACRO.RESet  ; Unregisters all GLOBALON commands and
              ; deletes all global PRACTICE macros
```

- To unregister just a selected **GLOBALON** command, type at the TRACE32 command line:

```
END ; Ends all active PRACTICE scripts

; Unregisters the action for the user-defined command TICONS
GLOBALON CMD TICONS ; Do not include the DO <action> here!
```

Result: The respective line or lines are no longer displayed in global PRACTICE stack frame of the **PMACRO.list** window. Thus the **GLOBALON** command or commands can no longer be executed.

See also

- DO
- ENTRY
- LOCAL
- ON

Format:	**GOSUB** *<subroutine>* [*<parameter_list>*]
<subroutine>:	*<name>* | *<label>*

The PRACTICE script continues at the defined **SUBROUTINE** or label. GOSUB can pass parameters to the subroutine. The subroutine can take over the parameters using commands **PARAMETERS** or **ENTRY.** The subroutine can return parameters using the command **RETURN**. The caller can take over the return parameters using **RETURNVALUES** or **ENTRY**.

NOTE: Recommendation for new scripts:
- define subroutines using **SUBROUTINE**
- pass parameters as string and take over using **PARAMETERS**
- return the result as string and take over using **RETURNVALUES**

Example using SUBROUTINE, PARAMETERS and RETURNVALUES:

```
SUBROUTINE initMem
(
  PRIVATE &address &memok
  PARAMETERS &address
  Data.Set &address++0x0FFF %Long 0x55AA55AA
  Data.Set &address++0x0FFF %Long 0x55AA55AA /DIFF
  &memok=!FOUND()
  RETURN "&memok"
)

GOSUB initMem "0x10000"
PRIVATE &ok
RETURNVALUES &ok
IF !&ok
  PRINT %ERROR "Mem init failed."
```

Example using label:

```
;     <label>       <parameter_list>
GOSUB mySubroutine1 0x100 10. "abc"
…
ENDDO


mySubroutine1:
     ENTRY    &address &len &string
     Data.Set &address++(&len-1) &string
     RETURN
```

NOTES:	• GOSUB accepts both labels and subroutine names as target, therefore labels and subroutines can not have the same name. • Labels must start in the first column of a line and end with a colon. No preceding white space allowed.

See also

- GOTO
- ON
- DO
- RETURN
- ENTRY
- JUMPTO

Format:	**GOTO** *<label>*

The PRACTICE script continues to execute at the **defined label**. You can also jump out of script blocks. In interactive mode, the PRACTICE command can be altered using the **GOTO** command.

Examples:

```
GOTO endloop                    ; label as jump destination

GOTO 102.                       ; line number as jump destination

&abc="lab10"                    ; PRACTICE macro for variable jump
GOTO &abc                       ; destination
```

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

See also

- GOSUB
- JUMPTO

▲ 'Release Information' in 'Legacy Release History'

IF — Conditional script execution

Format:	**IF** *<condition>*

The command or script block following an **IF** statement will be executed, if the condition is true. Emulators and debuggers have a counterpart of this command that works in the HLL syntax of the target program (command **Var.IF**). The syntax for *<condition>* is the same as for boolean command parameter.
For detailed information on booleans, refer to **"Parameter Types"** in PowerView User's Guide, page 35 (ide_user.pdf)

NOTE:	**IF** must be followed by a white space.

If the IF-body consists of just one line, then parentheses can be omitted:

```
IF   Register(d0)>0x0
     PRINT "Register not zero"
ELSE
     PRINT "Register zero"
```

```
Var.IF flags[5]>35              // HLL expression in condition
     STOP
ELSE
     Step
```

IF-bodies consisting of two or more lines must be enclosed in parentheses:

```
IF (Register(PC)!=ADDRESS.OFFSET(main))
(
  PRINT %ERROR "halted at wrong address!"
  ENDDO
)
```

Please avoid the following **mistake**:

```
IF Register(d0)==0x0  ( step )      ; not allowed
```

See also

- ELSE
- STATE.RUN()
- 'Release Information' in 'Legacy Release History'

Format:	**INKEY** [*<par>*]

The command waits for a keystroke. The code of the key is placed in the PRACTICE macro. If no parameter is used, the command stops script execution until any key is be pressed on the keyboard.

Examples:

```
INKEY                          ; wait until any key is pressed
```

```
INKEY &key                     ; wait for one key from keyboard
IF &key==0x0d                  ; check for carriage return
```

See also

- ENTER
- WAIT

▲ 'I/O Commands' in 'Training Script Language PRACTICE'

JUMPTO Global script jump

Format:	**JUMPTO** *<label>*

The PRACTICE script continues to execute at the **defined label**. Subroutine calls and **block nestings** are removed from the PRACTICE stack. The target label must reside in a stacked script module. The label must reside on the **top level** of a module!

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

See also

- DO
- GOSUB
- GOTO

Format:	**LOCAL** {*<macro>*}

The **LOCAL** command is used to create an empty PRACTICE macro in the current block. The macro hides any macro with the same name potentially created earlier (e.g. at a higher level of the PRACTICE stack).

The word *local* implies the **life-time** of the macro: it is created with the **LOCAL** command and erased when the declaring block is left.

PRACTICE macros declared with **LOCAL** are visible in all subsequently executed code within their life-time (unless hidden by later macro declarations). In particular they are visible in:

Yes Subroutines (**GOSUB** ...**RETURN**)

Yes Sub-scripts (**DO**...**ENDDO**)

Yes Sub-blocks (**IF**..., **RePeaT**, **WHILE**, etc.)

NOTE:	For creating macros only visible within their declaring block (as in the C programming language) use the command **PRIVATE**.

Example 1:

```
LOCAL &a &b &c
ENTRY &a &b
&c=&a*&b
ENDDO &c
```

Example 2:

```
LOCAL &result

GOSUB myfunc 2. 3.
ENTRY &result

PRINT "Result is &result"

ENDDO


myfunc:
     LOCAL &a &b &c
     ENTRY &a &b
     &c=&a*&b
     RETURN &c
```

Example 3: This script prints two sequences of the numbers 1, 2, 3, 4:

```
LOCAL &I   ;declare macro &I
&I=1.      ;assign value

WHILE &I<=2.
(
  PRINT "--Sequence #" &I "--"
  GOSUB sequence
  &I=&I+1.
)

ENDDO


sequence:
    LOCAL &I   ;hides the previously declared &I and declares new macro
    &I=1.
    WHILE &I<=4.
    (
      GOSUB output
      &I=&I+1.
    )
    RETURN

output:
    PRINT &I   ;uses the macro &I declared in subroutine 'sequence'
    RETURN
```

See also

- ENTRY
- GLOBAL
- GLOBALON
- PMACRO.EXPLICIT
- PMACRO.IMPLICIT
- PMACRO.list
- PRIVATE
- SPRINTF

ON Event-controlled PRACTICE script execution

[<events>] [<actions>] [Examples]

Format:	**ON** *<event>* [*<action>*]
<event>:	*<device_specific_events>* *<practice_specific_events>* *<cpu_specific_events>*
<practice_specific>:	**ALWAYS** **ERROR** **STOP** **CMD** *<command_name>* **TIME** *<delay>*
<action>:	**inherit** **CONTinue** **DO** *<file>* **GOSUB** *<label>* | *<block>* **GOTO** *<label>* | *<block>* **JUMPTO** *<label>* | *<block>* **DEFault**

The **ON** command enables the automatic start or branching of the PRACTICE scripts controlled by several events. The registered actions are stored on the PRACTICE stack, therefore the command is only valid in the block in which it was set, and in the subroutines called in this block. The currently active **ON** command can be viewed with the **PMACRO** command. The PRACTICE script will be started and stopped automatically, if the **GOSUB** action is used. If no target label is given, the line or block after the **ON** command will be executed instead.

NOTE: If you want the action to remain permanently active, use the **GLOBALON** command.

Using the **GLOBALON** command, you can create actions for global events, which are available for an entire TRACE32 session.

Events: <practice_specific_events> for the ON Command

Events	Descriptions
ALWAYS	The defined PRACTICE sequence is executed permanently, as long as no keyboard input occurs or no PRACTICE script is activated.
ERROR	Will be executed if a syntax or runtime error occurs in PRACTICE. The default action of this event is to halt script execution.
STOP	Will be executed, when the STOP button from the toolbar is pushed. Warning: A PRACTICE script using this feature may hang and cannot be stopped then by the STOP button!
CMD *<command_name>*	Definition of a new command. The TRACE32 commands can be extended with a user-defined command. The length of the *<command_name>* is limited to 9 characters and the character set [0..9], [@..Z], [a..z], '_', '+' and '-'. Arguments can be passed in the usual way, see **ENTRY**.
TIME *<delay>*	Executes the script after a delay time.

Events: <device_specific_events> for the ON command

<device_specific_events>	For a description of the device-specific events, refer to **ON** in the **General Commands Reference Guide O**.

Events: <cpu_specific_events> for the ON command

<cpu_specific_events>	Debuggers providing CPU-specific events are listed in the **See also** block at the end of the **ON** command description in the **General Commands Reference Guide O**.

<actions> for the ON Command

One of the following actions can be defined for any of the above events:

Actions	Descriptions
inherit	An already defined action for this event in the current stack level will be removed. If an action is registered in a higher stack level, the action of the higher stack level will be inherited. If no action is registered in any higher stack level, the debugger's default action will be performed (e.g. stop on error event).

Actions	Descriptions
CONTinue	If the event occurs, script execution will be continued. Use this option e.g. to ignore errors which would cause the script execution to halt.
DO	If the event occurs, the PRACTICE script in the specified file will be executed.
GOSUB	If the event occurs, a subroutine call will occur. The subroutine can be specified as a label, or inline as a PRACTICE block. With **RETURN**, the subroutine will return to normal script execution. For the **ERROR** event, the subroutine will return to the line after the command which caused the error.
GOTO	If the event occurs, the script execution will continue at the specified label, or in the specified inline PRACTICE block.
JUMPTO	If the event occurs, the script execution will continue at the specified label, or in the specified inline PRACTICE block. Subroutine calls and block nestings are removed from the PRACTICE stack. Use this action e.g. as global error/expection handler.
DEFault	The debugger's default action will be performed if the specified event occurs. Actions defined in a higher stack level for this event will be ignored. Use this action e.g. inside an event handler subroutine to avoid re-entry while the handler's subroutine is active.

Example 1: Error Handler

```
ON ERROR JUMPTO errorhandler_filenotfound      ;set up error handler
Data.LOAD.ELF project.x
ON ERROR inherit                               ;disable error handler
…
ENDDO

errorhandler_filenotfound:
     PRINT %ERROR "File not found!"
     ENDDO
```

Example 2: Define a New Command

```
; define new command LoadEx <file>
ON CMD LoadEx GOSUB
(
     LOCAL &filename                              ;declare local macros
     ENTRY &filename                              ;get parameter(s)
     Break.Delete
     Data.LOAD.ELF "&filename"
     RETURN
)
STOP
```

Example 3: Timeout on User Input

To try out this script, simply copy it to a `test.cmm` file, and then run it in TRACE32 (See "**How to...**").

```
AREA.Create USRINP                      ;create and show message area
AREA.Select USRINP
AREA.view   USRINP
PRINT "Press return (within 5 seconds) to abort configuration > "

ON TIME 5.0s GOTO no_timeout            ;set up timeout
ENTER &invalue                          ;script waits here for user input

ON TIME 5.0s inherit                    ;disable timeout
PRINT "Configuration aborted."
ENDDO

no_timeout:
     PRINT "Configuration starting..."
     ENDDO
```

Example 4: Background Task in PRACTICE:

```
ON ALWAYS GOSUB
(
     PRINT Register(PC)
     RETURN
)
```

Example 5

A complex demo script is included in your TRACE32 installation. To access the script, run this command under Windows and Unix:

```
B::CD.PSTEP ~~/demo/practice/event_controlled_program/dialog_ontime.cmm
```

See also

- DO
- GLOBALON
- GOSUB

Format:	**OPEN #**<*buffer_number*> <*file*> [**/**<*option*>]
<*option*>:	**Read** | **Write** **Create** | **Append** **Binary**

Opens a file for reading or writing.

Write **Append**	When a file is opened in write mode, the data is per default appended at the end of the file (same behavior as with option **/Append**).
Create	Creates a new file and overwrites an old one if the file already exists. The default file format is text.
Create and **Binary**	The options **/Create /Binary** create a binary file.

Example:

```
OPEN  #1 datafile.dat /Read
READ  #1 %LINE &data
…
CLOSE #1
```

See also

- ■ APPEND
- ■ WRITEB
- ❑ FILE.EXIST()
- ❑ OS.FILE.readable()
- ■ CLOSE
- ■ Data.WRITESTRING
- ❑ FILE.OPEN()
- ❑ OS.FILE.SIZE()
- ■ READ
- ■ Var.WRITE
- ❑ OS.FILE.DATE()
- ■ WRITE
- ❑ FILE.EOF()
- ❑ OS.FILE.PATH()

▲ 'Release Information' in 'Legacy Release History'

PARAMETERS — Parameter fetching

Format: **PARAMETERS** {*<macro>*}

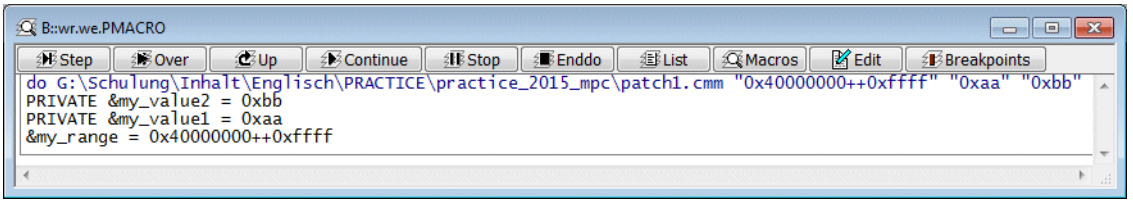
Fetches the parameters passed by PRACTICE script/subroutine calls.

Example 1: Parameters have to be enclosed in quotes (") in the call.

```
PSTEP patch1.cmm "0x40000000++0xffff" "0xaa" "0xbb"
```

The **PARAMETERS** command first creates **PRIVATE** macros for all macro names not found in the current scope (please be aware that the **LOCAL** macros of the caller(s) are always out of scope for the **PARAMETERS** command) and then assigns the passed values to the macros.

```
LOCAL &my_range
PARAMETERS &my_range &my_value1 &my_value2
```



Example 2:

```
GOSUB square "0x5" "5" "5."
ENDDO

square:
PARAMETERS &x &y &z       ;fetch parameters passed by the subroutine call

   PRINT &x*&x            ;result is printed as a hex value
   PRINT &y*&y            ;result is printed as a hex value
   PRINT %Decimal &y*&y   ;result is printed as a decimal value
   PRINT &z*&z            ;result is printed as a decimal value
   RETURN
```

See also

- ENTRY
- RETURNVALUES

▲ 'Release Information' in 'Legacy Release History'

Using the **PBREAK** command group, you can set, enable, disable, save, and clear *an unlimited number of* breakpoints in PRACTICE script files (*.cmm).

These program breakpoints are set with the **PBREAK.Set** command. The **PBREAK.List** window shows an overview of the breakpoints you have set in your PRACTICE script files (*.cmm). For compatibility reasons, TRACE32 continues to support the breakpoint that can be set with the deprecated **PBREAK.at** command.

See also

- PBREAK.Delete
- PBREAK.DISable
- PBREAK.ENable
- PBREAK.List
- PBREAK.OFF
- PBREAK.ON
- PBREAK.RESet
- PBREAK.Set
- PEDIT
- PLIST
- PSTEP
- CONTinue

PBREAK.Delete Delete breakpoint

Format: **PBREAK.Delete** [*<line>* [*<file>*]]

If no argument is passed, this command deletes all program breakpoints from the list displayed in the **PBREAK.List** window.

You can delete specific breakpoints by passing the following arguments:

<line>	Line number. Deletes all breakpoints at *<line>* for all PRACTICE scripts.
, *<file>*	Line number is omitted; path and file name of the PRACTICE script are specified. Deletes all breakpoints for the PRACTICE script *<file>*.
<line> *<file>*	Deletes the breakpoint at *<line>* for the PRACTICE script *<file>*.

See also

- PBREAK
- PBREAK.RESet

▲ 'Release Information' in 'Legacy Release History'

PBREAK.DISable Disable breakpoint

Format: **PBREAK.DISable** [*<line>*] [*<file>*]

If no argument is passed, this command disables all program breakpoints. Disabled breakpoints are flagged with a small gray bar in the **PLIST** and **PSTEP** windows.

You can disable specific breakpoints by passing the following arguments:

<line>	Line number. Disables all breakpoints at *<line>* for all PRACTICE scripts.
, *<file>*	Line number is omitted; path and file name of PRACTICE script are specified. Result: All breakpoints for the PRACTICE script *<file>* are disabled.
<line> <file>	Disables the breakpoint at *<line>* for the PRACTICE script *<file>*.

See also

- PBREAK
- PBREAK.ENable

▲ 'Release Information' in 'Legacy Release History'

PBREAK.ENable — Enable breakpoint

Format:	**PBREAK.ENable** [*<line>*] [*<file>*]

If no argument is passed, this command enables all program breakpoints. Enabled breakpoints are flagged with a small red bar in the **PLIST** and **PSTEP** windows.

You can enable specific breakpoints by passing the following arguments:

<line>	Line number. Enables all breakpoints at *<line>* for all PRACTICE scripts.
, *<file>*	Line number is omitted; path and file name of PRACTICE script are specified. Result: All breakpoints for the PRACTICE script *<file>* are enabled.
<line> <file>	Enables the breakpoint at *<line>* for the PRACTICE script *<file>*.

See also

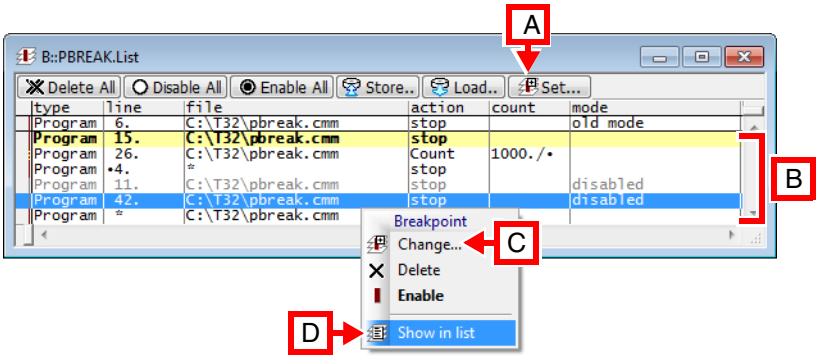
- PBREAK
- PBREAK.DISable

▲ 'Release Information' in 'Legacy Release History'

Format:	**PBREAK.List**

Lists all program breakpoints you have created for PRACTICE script files (*.cmm).

The **PBREAK.List** window shown below provides the following commands via the toolbar and the popup menu: You can delete, disable, and enable all breakpoints or a selected breakpoint. In addition, you can save the breakpoints to file with the **STOre** command, and load them for the next session with the **DO** command.



A Set a new breakpoint.

B Breakpoints created with **PBREAK.Set**.

C Edit the selected breakpoint.

D Open the file in the **PLIST** window at line number 42.

Descriptions of Columns in the PBREAK.List Window:

- **type**: PRACTICE supports program breakpoints. **Yellow** indicates that script execution has stopped at this breakpoint. You can continue script execution with the **CONTinue** command.
- **line**: Line number of breakpoint
  - An asterisk (*) in the **line** column indicates that a breakpoint is set at the first *executable* line in this PRACTICE script.
  - A dot before a line number indicates that the breakpoint was originally set to an empty line or comment line. Such a breakpoint automatically moves to the next executable line.
- **file**: Lists all PRACTICE script files where breakpoints have been set.
  - An asterisk (*) in the **file** column indicates that a breakpoint is dynamically set in any active PRACTICE script at the specified line number (see **PBREAK.Set ... /AnyFile**). This is useful for testing nested PRACTICE script files.
- **action**: Action of the breakpoint, i.e. **stop** or **Count**. See **PBREAK.Set ... /Count**.
- **count**:
  - Counts how often the PRACTICE command has hit the counter breakpoint. Counter breakpoints are typically set at counters in **WHILE** loops.
  - A dot after a counter indicates that the counter limit is unspecified. See **PBREAK.Set ... /Count**.

- **mode**:
  - **Old mode**: This breakpoint was created with the deprecated **PBREAK.at** command.
  - **Disabled** is displayed for disabled breakpoints. All other breakpoints are enabled.
  - **Temporary** is displayed until the PRACTICE command has reached the line with the temporary breakpoint.
    A temporary breakpoint is created when you right-click a script line in a **PLIST** window and select **Go Till**.

See also

■ PBREAK

▲ 'Release Information' in 'Legacy Release History'

PBREAK.OFF TRACE32 disables breakpoint handling

Format:	**PBREAK.OFF**

Disables all breakpoints for the script currently loaded on top of the PRACTICE stack. If no PRACTICE script is running this command is locked.

TRACE32 generates a breakpoint script when you click the **Store** button in a **PBREAK.List** window. In this script, your breakpoint settings are enclosed between the **PBREAK.OFF** and **PBREAK.ON** command. The two commands ensure that a breakpoint script can be re-loaded without being stopped by the breakpoints it contains.

Example:

```
PBREAK.RESet                      ;reset all PRACTICE breakpoints
PBREAK.OFF                        ;switch off further breakpoint checking

PBREAK.Set 5. , , /AnyFile        ;set a breakpoint to line 5 in any file
PBREAK.Set 8. c:\t32\test.cmm     ;set a breakpoint to line 8 in file

PBREAK.ON                         ;switch on breakpoint checking again
```

See also

■ PBREAK.ON ■ PBREAK

PBREAK.ON TRACE32 re-enables breakpoint handling

Format:	**PBREAK.OFF**

Re-enables all breakpoints for the script currently loaded on top of the PRACTICE stack. If no PRACTICE script is running this command is locked.

TRACE32 generates a breakpoint script when you click the **Store** button in a **PBREAK.List** window. In this script, your breakpoint settings are enclosed between the **PBREAK.OFF** and **PBREAK.ON** command. The two commands ensure that a breakpoint script can be re-loaded without being stopped by the breakpoints it contains.

See also

- PBREAK.OFF
- PBREAK

PBREAK.RESet Clear all breakpoints

Format:	**PBREAK.RESet**

Deletes all breakpoints from the list displayed in the **PBREAK.List** window.

See also

- PBREAK
- PBREAK.Delete

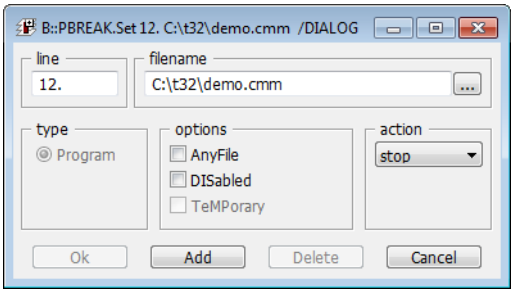
[Examples]

Format:	**PBREAK.Set** *<line>* *<file>* [**/DISabled** | **/Count** | **/AnyFile** | **/DIALOG**] **PBREAK** [*<line>*] [*<file>*] (deprecated) **PBREAK.at** [*<line>*] [*<file>*] (deprecated)

Sets a program breakpoint at *<line>* for the PRACTICE script *<file>*. Optionally, the **PBREAK.Set** dialog can be displayed. If the specified line number refers to an empty line or a PRACTICE comment, then the breakpoint is set at the next executable line.

By default, a program breakpoint in a PRACTICE script stops script execution. However, if a program breakpoint is modified with the **Count** option, script execution continues, and breakpoint hits are counted.

The **PBREAK.List** window lists the breakpoints you have created with **PBREAK.Set**.



, *<file>*	Line number is omitted; path and file name of PRACTICE script are specified. Result: A breakpoint is set at the first executable line in the PRACTICE script *<file>*.
<line>	Line number. Sets a breakpoint at *<line>* in the current PRACTICE script or in any script if none is loaded.
<line> *<file>*	Sets a breakpoint at *<line>* in the PRACTICE script *<file>*.
AnyFile	Sets a breakpoint at *<line>* in any PRACTICE script file. You need to replace *<file>* with two **,** **,** (commas) when using **AnyFile**. TRACE32 then applies the breakpoint to the currently active PRACTICE script. **AnyFile** is intended for testing nested PRACTICE script files. It automatically stops script execution at the first executable line of any PRACTICE script file that is called next. See example.
Count	The breakpoint is created as a counter which does not stop the execution of a PRACTICE script, but counts breakpoint hits. Useful for counters in **WHILE** loops. See example.

DIALOG	Displays the **PBREAK.Set** dialog. Clicking the **Add** button closes the dialog and makes the PRACTICE breakpoint available in TRACE32. Available breakpoints are listed in the **PBREAK.List** window. See example.
DISabled	The breakpoint is created as a disabled breakpoint.

Example for the Count option: The following example is for demo purposes only. To try this script, copy it to a `test.cmm` file, and then run it in TRACE32 (See "**How to...**").

```
PBREAK.Set 7. , , /Count    ;set a breakpoint in line 7. as loop counter
PBREAK.List                 ;display the list of breakpoints set in
                            ;PRACTICE script files
&tmp=0.
WHILE &tmp<1000.
(
   &tmp=&tmp+1.             ;the spot breakpoint is set here
                            ;note that the 'count' column in the
                            ;PBREAK.List updates automatically

   PRINT "looprun: &tmp"    ;prints the counter to the message bar
)

PRINT "loop finished"       ;right-click in the PLIST window, and then
                            ;select 'Go Till'

PBREAK.Delete 7.            ;delete all breakpoints at line 7. for
                            ;all PRACTICE scripts!
```

Example for the AnyFile option:

```
PBREAK.List                 ;display breakpoint list for PRACTICE scripts
PBREAK.Set 1. , , /AnyFile  ;set breakpoint to first executable line
                            ;in any file

CD.DO file1.cmm             ;script execution automatically stops at the
;       |___ file2.cmm      ;first executable line of each file
;              |___ file3.cmm

PBREAK.RESet   ;always clear all breakpoints at the end of your session
```

Example for the DIALOG option:

```
;opens the list of PRACTICE breakpoints
PBREAK.List

;sets a breakpoint in line 12. of the demo.cmm file and
;opens the PBREAK.Set dialog
PBREAK.Set 12. c:\t32\demo.cmm  /DIALOG
;click the Add button:
;- the dialog closes
;- the PRACTICE breakpoint appears in the list of PRACTICE breakpoints


;sets a breakpoint in line 49. for the currently active script file and
;opens the PBREAK.Set dialog
PBREAK.Set 49. , ,   /DIALOG
;click the Add button:
;- the dialog closes
;- the PRACTICE breakpoint appears in the list of PRACTICE breakpoints


;the * lets you select a script file from the file browser,
;then the PBREAK.Set dialog opens
PBREAK.Set , * /DIALOG
;click the Add button:
;- the dialog closes
;- the PRACTICE breakpoint appears in the list of PRACTICE breakpoints
```

See also

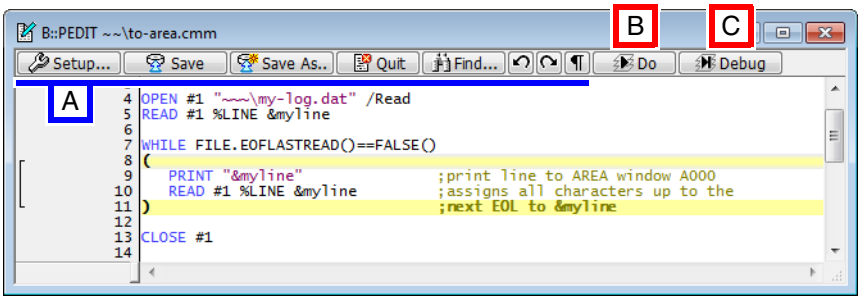
■ PBREAK

▲ 'Release Information' in 'Legacy Release History'

[Examples]

Format:	**PEDIT** *<file>* [*<line>*] [*/<option>*]
<option>:	**AutoSave** | **NoSave**

Opens the TRACE32 editor window **PEDIT**, where you can create and edit PRACTICE script files (*.cmm). The editor window provides syntax highlighting, configurable auto-indentation as well as multiple undo and redo. **PEDIT** works in the same manner as the **EDIT** command.



Buttons common to all TRACE32 editors:

A For button descriptions, see **EDIT.file**.

Buttons specific to this editor:

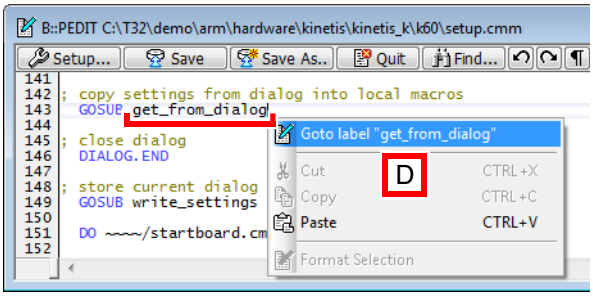
B Execute the PRACTICE script (**DO**).

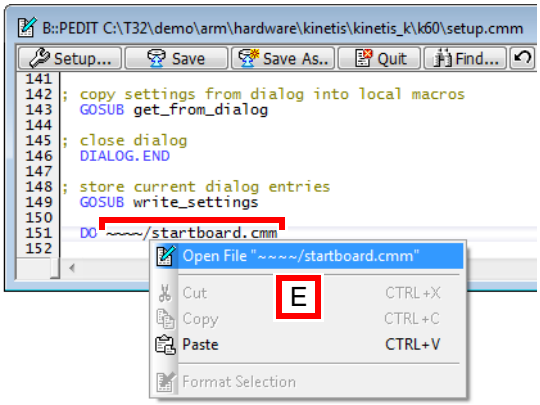
C Debug the PRACTICE script in a **PLIST** window.

<file>	The default extension for *<file>* is ***.cmm**.
<line>, *<option>*	For description of the arguments, see **EDIT.file**.

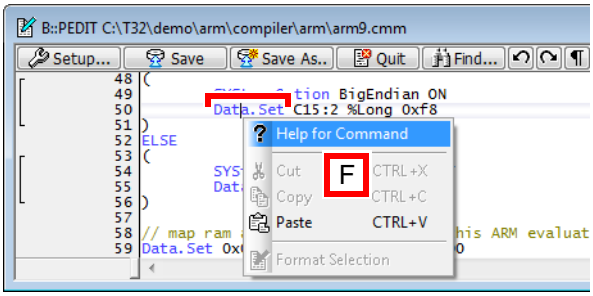
Tips

Tip 1: You can quickly jump to the destination of a label by right-clicking a label and selecting **Goto Label** from the popup menu [**D**]. A similar feature exists for files [**E**]. These navigation features are useful in very long PRACTICE script files.





Tip 2: You can quickly get help on an unknown command or function by right-clicking the command or function. Then select **Help for Command** or **Help for Function** from the popup menu [**F**].



Examples

Example 1:

```
;open the file 'test.cmm' and place the insertion point in line 115
PEDIT test.cmm 115.
```

Example 2:

```
;- The WinExt pre-command allows you to move the PEDIT window outside the
;  TRACE32 main window.
;- The WinResist pre-command prevents the PEDIT window from being closed
;  by the WinCLEAR command in your PRACTICE scripts.
WinExt.WinResist.PEDIT test.cmm
```

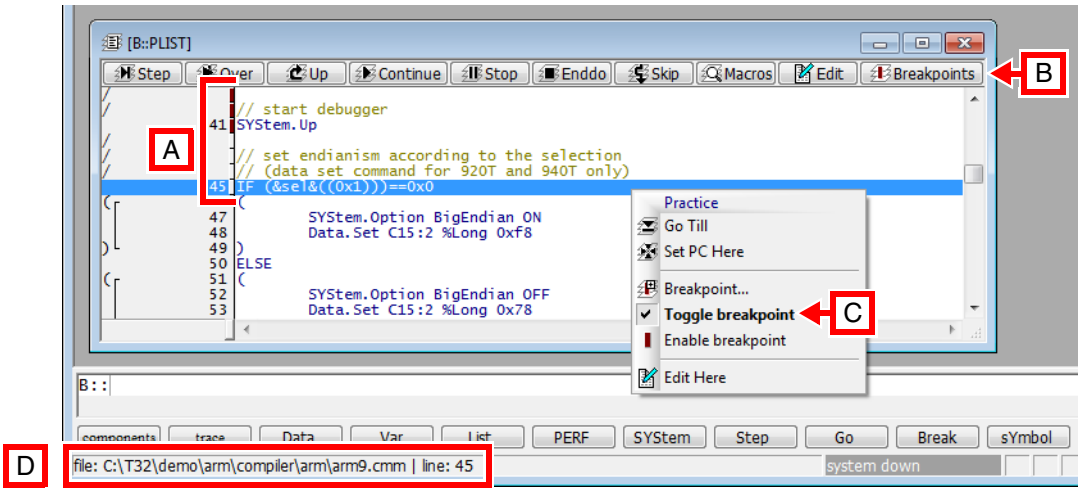
See also

- PBREAK
- PLIST
- PSTEP
- ClipSTOre
- EDIT.file
- SETUP.EDITOR

▲ 'Text Editors' in 'PowerView User's Guide'
▲ 'Release Information' in 'Legacy Release History'
▲ 'Create a PRACTICE Script' in 'Training Script Language PRACTICE'

Format:	**PLIST** [*<label>* | *<line>*]

Lists the currently loaded PRACTICE script. Script nesting, branching and single jumping are marked in the scale area on the left. Enabled breakpoints are flagged with a small red bar, disabled breakpoints are flagged with a small gray bar in the **PLIST** window.



A An enabled breakpoint in line 41 at SYStem.Up; a disabled breakpoint in line 45 at the IF block.

B Opens the PBREAK.List window, listing the PRACTICE breakpoints.

C Double-clicking a line toggles a breakpoint (set/delete).

D Click inside the **PLIST** window to display the file name of the script being executed. The file name is displayed in the state line of the TRACE32 main window.

Examples:

```
PLIST 10.                       ; List as of line 10

PLIST myLabel                   ; List as of label 'myLabel'

PLIST                           ; List as of the current program counter

; an example for PLIST
PSTEP                           ; Set PRACTICE script execution to
                                ; single step mode
DO test.cmm                     ; Load PRACTICE script
PLIST                           ; List PRACTICE script for debugging
```

See also

■ PBREAK ■ PEDIT ■ PMACRO ■ PSTEP

▲ 'Release Information' in 'Legacy Release History'

If a PRACTICE script is executed, the PRACTICE stack frame and the PRACTICE macros (variables) can be visualized and the macro search priority can be manipulated by the commands of the **PMACRO** command group.

See also

- PMACRO.EXPLICIT
- PMACRO.IMPLICIT
- PMACRO.IMPLICITPRIVATE
- PMACRO.list
- PMACRO.LOCK
- PMACRO.RESet
- PMACRO.UNLOCK
- PLIST

▲ 'Release Information' in 'Legacy Release History'

PMACRO.EXPLICIT Enforce explicit PRACTICE macro declaration

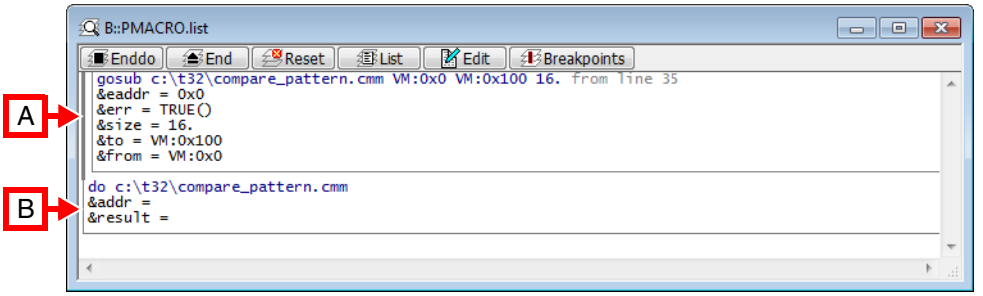
[Examples - GLOBAL macros] [Examples - LOCAL macros]

Format:	**PMACRO.EXPLICIT**

Enforces explicit declarations of PRACTICE macros. That is, macros in PRACTICE scripts (*.cmm) must be declared with the commands **GLOBAL** or **LOCAL** or **PRIVATE**.

Starting a PRACTICE script with **PMACRO.EXPLICIT** enforces an explicit declaration of all PRACTICE macros *for the entire PRACTICE script*. The explicit declaration in the parent PRACTICE script extends to all blocks, sub-blocks (e.g. **IF**..., **RePeaT**..., **WHILE**..., etc.), subroutines (**GOSUB**...**RETURN**), and sub-scripts (**DO**...**ENDDO**).

Starting a sub-block or subroutine with **PMACRO.EXPLICIT** enforces explicit declarations *only for the sub-block* (e.g. **IF**..., **RePeaT**..., **WHILE**..., etc.) or *subroutine* (**GOSUB**...**RETURN**).



A Thick gray bar = explicit macro declaration range; here in a subroutine block (**GOSUB**...**RETURN**).

B Thin gray bar = implicit macro declaration range.

If the explicit-declaration setting is violated during PRACTICE script execution, the TRACE32 message line informs you with an error message and script execution stops.

- If an undeclared macro is initialized, the following error message is displayed in the state line: “explicit PRACTICE macro declaration expected”.
- If an explicitly declared macro is declared again, the following error message is displayed in the state line: “explicitly declared PRACTICE macro already exists”.

NOTE:	• Explicit macro declaration is recommended for fail-safe operation of PRACTICE scripts. • **PMACRO.EXPLICIT** together with the error messages assists you in declaring all macros explicitly.

Examples - GLOBAL Macro

Example 1: When **GLOBAL** macros are declared *before* the **PMACRO.EXPLICIT** command is executed, then the same **GLOBAL** macros can be re-declared at any time during a TRACE32 session (but not while the parent script is still running). The parameter of a **GLOBAL** macro can re-initialized by any PRACTICE script.

```
GLOBAL &ProjName1        ;declare PRACTICE macros with GLOBAL
GLOBAL &Start1

PMACRO.EXPLICIT          ;enforce explicit PRACTICE macro declaration

&ProjName1="Project 10" ;initialize the GLOBAL macros
&Start1=CLOCK.TIME()
```

Example 2: When **GLOBAL** macro declarations are placed *after* **PMACRO.EXPLICIT**, the **GLOBAL** macros can be declared only once during a TRACE32 session. If they are declared again, a script error occurs, script execution stops, and an error message is displayed. The parameter of a **GLOBAL** macro can re-initialized by any PRACTICE script.

```
PMACRO.EXPLICIT        ;enforce explicit PRACTICE macro declaration

GLOBAL &ProjName2      ;declare PRACTICE macro explicitly with GLOBAL
GLOBAL &Start2

&ProjName2="Poject X" ;initialize the GLOBAL macro
&Start2=CLOCK.TIME()
```

Example 3: Correct explicit declaration of the PRACTICE macro **`LOCAL &file1`**

```
PMACRO.EXPLICIT   ;enforce explicit PRACTICE macro declaration

LOCAL &file1      ;declare PRACTICE macro explicitly with LOCAL

&file1=OS.PresentExecutableFile()   ;initialize PRACTICE macro

PRINT "&file1"    ;print name of the currently used TRACE32 executable
```

Example 4: The explicit declaration of the PRACTICE macro **`LOCAL &file2`** is missing, which will result in an error message.

```
PMACRO.EXPLICIT   ;enforce explicit PRACTICE macro declaration

;here, the explicit declaration with the LOCAL command is missing

&file2=OS.PresentExecutableFile()
                  ;now, a script error occurs, script execution stops,
                  ;and the following error message is displayed:
                  ;"explicit PRACTICE macro declaration expected"
```

See also

- PMACRO
- PMACRO.IMPLICIT
- PRIVATE
- GLOBAL
- LOCAL

▲ 'Release Information' in 'Legacy Release History'

PMACRO.IMPLICIT — Implicit PRACTICE macro declaration

Format:	**PMACRO.IMPLICIT**

Ends the explicit macro declaration range started with the **PMACRO.EXPLICIT** command.

NOTE:	By default, TRACE32 declares macros implicitly if you neglected to declare them with the commands **LOCAL** or **PRIVATE**.

See also

- PMACRO
- PMACRO.EXPLICIT
- PRIVATE
- GLOBAL
- LOCAL

▲ 'Release Information' in 'Legacy Release History'

Format:	**PMACRO.IMPLICITPRIVATE**

Per default, macros implicitly created by an assignment are visible in sub-functions and scripts called with **DO**. After calling the new command **PMACRO.IMPLICITPRIVATE**, these implicitly created macros are hidden in sub-functions and scripts called with **DO**. So, these macros behave as if they were created with the **PRIVATE** command.

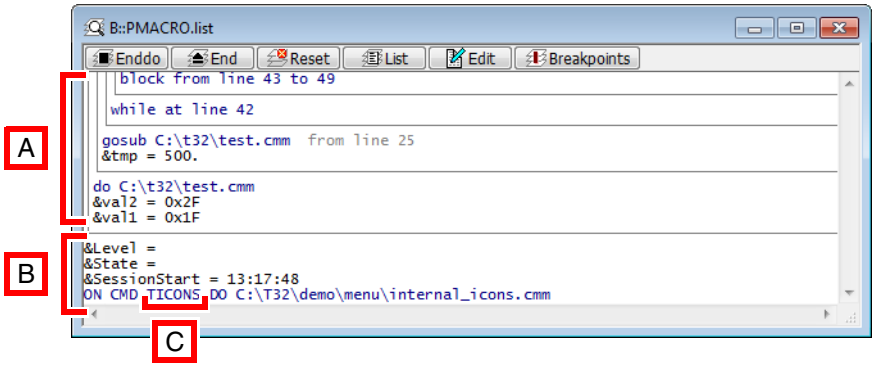
See also

- PMACRO

Format:	**PMACRO.list**

Displays the global, local, and private PRACTICE macros. In addition, the **PMACRO.list** window displays:

- The global PRACTICE stack frame
- Local PRACTICE stack frames
- Script nesting
- Loop commands (**WHILE**, **RePeaT**)
- User-defined **ON** and **GLOBALON** commands



A Local PRACTICE stack frames

B Global PRACTICE stack frame

C Example of a user-defined command named TICONS

See also

- PMACRO
- PRIVATE
- END
- GLOBAL
- LOCAL

PMACRO.LOCK Lock PRACTICE macros

Format:	**PMACRO.LOCK**

All macros of higher nesting levels are disabled. The lock will be released automatically if the current level is left.

Example:

```
LOCAL &m1

&m1="Hello"
PRINT "main: m1 = &m1"
GOSUB ltest World
PRINT "main: m1 = &m1"
ENDDO

; -----------------------------------

ltest:
(
  PMACRO.LOCK
  (
   ENTRY &m1
   PRINT "ltest: m1 = &m1"
  )
  PMACRO.UNLOCK
  RETURN
)
```

See also

- PMACRO
- PMACRO.UNLOCK

PMACRO.RESet — Clear current PRACTICE macros

Format:	**PMACRO.RESet**

All macros in the current level are removed. If no PRACTICE script is running, all macros on the top level will be erased.

See also

- PMACRO
- END
- GLOBAL
- GLOBALON

Format:	**PMACRO.UNLOCK**

Re-enables the macros that were previously disabled with **PMACRO.LOCK**.

See also

- PMACRO
- PMACRO.LOCK

[Examples]

Format:	**PRINT** [{%*<attribute>*}] [{ {%*<format>*} *<data>* }]
<attribute>:	**AREA** *<area_name>* **CONTinue** **HOME**
<format>:	*<type>* **COLOR.***<color>* **ERROR** **WARNING**
<type>:	**Ascii** | **BINary** | **Decimal** | **Hex** | **String**
<color>:	**NORMAL** | **BLACK** | **MAROON** | **GREEN** | **OLIVE** | **NAVY** | **PURPLE** **TEAL** | **SILVER** | **GREY** | **RED** | **LIME** | **YELLOW** | **BLUE** | **FUCHSIA** **AQUA** | **WHITE**

Writes the given arguments to the selected **AREA** window. When writing to the default **AREA A000**, the written data is also shown in the TRACE32 message line.

What is the difference between the commands ...?

PRINT	**ECHO**
Writes all data without any format decoration (e.g. without the prefix “0x” for hexadecimal numbers).	Writes all data decorated to indicate the format of the data.
For a comparison of the different outputs, see example 4.	

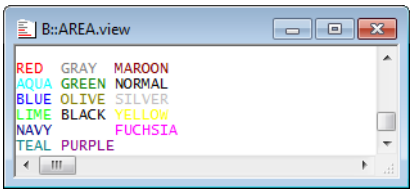
<type>	Converts all following *<data>* to the specified format. See example 3.
AREA *<area_name>*	Writes the message to the specified message area. Without this attribute the message is written to the default **AREA A000** or the message area that has just been selected with the command **AREA.Select**. If the message AREA you want to write to does not yet exist, use the command **AREA.Create**. This attribute can only be used before the first *<data>*. (The execution of a single **PRINT** command writes only to one message area.)

CONTinue	Adds the string to the current output line in the selected **AREA** window or TRACE32 message line without inserting a newline character. See example 2. This attribute can only be used before the first *<data>*.
COLOR.*<color>*	Writes *<data>* in color in the **AREA** window. See example 1.
ERROR	*<data>* will be written in dark red. Only when writing to the default **AREA** **A000**, the text is also shown as an error in the TRACE32 message line.
WARNING	*<data>* will be written in orange. Only when writing to the default **AREA** **A000**, the text is also shown as a warning in the TRACE32 message line.
HOME	Writes to top line in **AREA** window. This attribute can only be used before the first *<data>*.

Example 1

An overview of the available colors and their names is written to the **AREA** window:

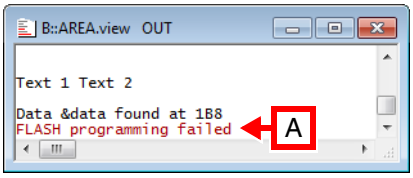
```
AREA.view

PRINT %COLOR.RED   "RED   " %COLOR.GRAY    "GRAY   " %COLOR.MAROON   "MAROON"
PRINT %COLOR.AQUA  "AQUA  " %COLOR.GREEN   "GREEN  " %COLOR.NORMAL   "NORMAL"
PRINT %COLOR.BLUE  "BLUE  " %COLOR.OLIVE   "OLIVE  " %COLOR.SILVER   "SILVER"
PRINT %COLOR.LIME  "LIME  " %COLOR.BLACK   "BLACK  " %COLOR.YELLOW   "YELLOW"
PRINT %COLOR.NAVY  "NAVY  " %COLOR.WHITE   "WHITE  " %COLOR.FUCHSIA  "FUCHSIA"
PRINT %COLOR.TEAL  "TEAL  " %COLOR.PURPLE  "PURPLE"
```



Example 2

Various messages are written to a new, user-defined **AREA** window named OUT:

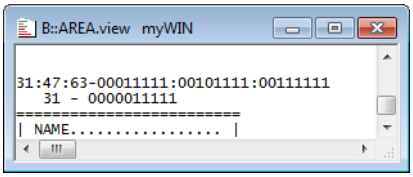
```
;define position, size and name of the new AREA window named OUT
WinPOS 3.5  10.  80.  6.  0.  0.  OUT
AREA.Create OUT                        ;create AREA window OUT

AREA.Select OUT                        ;select the AREA window OUT
                                       ;for printing

AREA.view   OUT                        ;display AREA window OUT
PRINT "Text 1 "
PRINT %CONTinue "Text 2 "              ;print text without line break
PRINT ""                               ;print an empty line

                                       ;print user-defined error message
PRINT "Data &data found at " ADDRESS.OFFSET(TRACK.ADDRESS())
PRINT %ERROR "FLASH programming failed"

;WinClear OUT                          ;remove AREA window OUT
```



A This user-defined error message is *not* written to the TRACE32 message line, since we have re-routed the output to the user-defined **AREA** window named OUT.

Example 3

```
;print hex values in decimal and then in binary format
PRINT %Decimal 0x1f ":" 0x2f ":" 0x3f "-" %BINary 0x1f ":" 0x2f ":" 0x3f

;print all data in certain format using PRACTICE formatting functions
PRINT FORMAT.Decimal(5.,0x1f)  " - "  FORMAT.BINary(10.,0x1f)

;formatted printing (includes character repetition with <<)
PRINT "="<<24.
PRINT "| " "NAME"  "."<<(20.-STRing.LENgth("NAME")) " |"
```



Example 4

The following table shows the output in a message area for the same data written with **ECHO** and **PRINT**:

<data>	AREA output with ECHO	AREA output with PRINT
0x042	`0x42`	`42`
%Hex 66.	`0x42`	`42`
23.	`23.`	`23`
%Decimal 0x17	`23.`	`23`
0y110011	`0y00110011`	`110011`
%BINary 0x33	`0y00110011`	`00110011`
'X'	`'X'`	`X`
%Ascii 0x58	`'X'`	`X`
5==5	`TRUE()`	`TRUE`
5==3	`FALSE()`	`FALSE`
"text"	`text`	`text`
P:0x001000	`P:0x1000`	`P:0x1000`
500ms	0.5000000000s	0.5000000000s
DATE.MakeUnixTime(1990.,10.,3,0,0,0)	654912000.	654912000
Var.VALUE(23 * 47)	0x439	439

See also

- PRINTF
- PRinTer
- AREA.CLEAR
- AREA.Create
- AREA.SAVE
- AREA.Select
- AREA.view
- ECHO
- SPRINTF
- WinPrint

▲ 'Release Information' in 'Legacy Release History'
▲ 'I/O Commands' in 'Training Script Language PRACTICE'

[Examples]

Format:	**PRINTF** [{%*<attribute>*}] "*<format_string>*" [{*<data>*}]
<attribute>:	**AREA** *<area_name>* **CONTinue** **COLOR.***<color>* **ERROR** **WARNING**
<color>:	**NORMAL** | **BLACK** | **MAROON** | **GREEN** | **OLIVE** | **NAVY** | **PURPLE** **TEAL** | **SILVER** | **GREY** | **RED** | **LIME** | **YELLOW** | **BLUE** | **FUCHSIA** **AQUA** | **WHITE**

Writes text and data to the default **AREA A000** or the selected **AREA** window in the style of the printf() function of C/C++. When writing to the default **AREA A000**, the output is also shown in the TRACE32 message line.

<attribute>	For a description of the attributes, see table below.
<format_string>	The characters of the *<format_string>* are written to a message area. However the following control characters within the *<format_string>* have a special meaning: • **&** (ampersand) • **"** (double-quotes) • **** (backslash) • **%** (percent sign) For information about the control characters, see table below.
<data>	One *<data>* argument is required for each control sequence started by a **%** within the *<format_string>*. See example. For information about the syntax of the control sequence, click here.

<attribute>	**Description**
AREA *<area_name>*	Writes the message to the specified message area. Without this attribute the message is written to the default **AREA A000** or the message area that has just been selected with the command **AREA.Select**. If the message AREA you want to write to does not yet exist, use the command **AREA.Create**.
CONTinue	Adds the message to the last output line in the currently selected **AREA** window.
COLOR.*<color>*, **ERROR**, **WARNING**	For option descriptions, see **PRINT**.

Control Characters	Description
& (ampersand)	Like in any other string in TRACE32, the ampersand invokes a text-replacement with a named PRACTICE macro. To safely output an ampersand write **`&"+"`** `PRINTF "Jekyll&"+"Hyde"` is printed as `Jekyll&Hyde`
" (double-quotes)	Like in any other string in TRACE32, a double-quote ends a string unless you escape it with another double-quote. `PRINTF "She said ""Hi!"""` is printed as `She said "Hi!"`
**** (backslash)	Unlike in C/C++ **the backslash is not a special escape character** inside TRACE32 strings. A backslash inside the *<format_string>* of **PRINTF** is printed like a regular character. `PRINTF "north\northwest"` is printed as `north\northwest` **NOTE:** Every **PRINTF** command will automatically cause a line break in the **AREA** window before writing to it. You can explicitly suppress the line break with the attribute **%CONTinue**, which has to be placed outside the *<format_string>*. (See **%CONTinue**)
% (percent sign)	The percent sign is the magic control character with **PRINTF**: Any percent sign (**%**) inside the *<format_string>* starts a control sequence which is replaced by the *<data>* arguments following the *<format_string>*. For every valid control sequence, you must specify a *<data>* argument. The control sequence started with the percent sign has the form: • **%**[flags][width][**.**precision][length]specifier The percent sign and specifier are mandatory; the rest is optional. The length is highly optional and mainly accepted for compatibility to the format-string in C/C++. For information about the available flags, specifiers, etc., click the blue hyperlinks. To print a percent sign, you have to write **`%%`** `PRINTF "100%% safe"` is printed as `100% safe`

Specifier	Description	Output	Valid Argument Types

d or **i**	Signed decimal integer
	`PRINTF "%i" -314.159265`
	`PRINTF "%i" -0x13A ;minus sign`
	`PRINTF "%i" ~0x13A+1 ;tilde`
	`PRINTF "%hi" 0xFEC6`
	For information about the **h**, see Length.
		-314	Hex, Decimal, Binary, Boolean, ASCII, Address, Float, Time*

u	Unsigned decimal integer	314	
o	Unsigned octal integer	472	
x	Unsigned hexadecimal integer, lowercase digits	13a	
X	Unsigned hexadecimal integer, uppercase digits	13A	
!y	Unsigned binary integer (TRACE32 extension)	100111010	Hex, Decimal, Binary, Boolean, ASCII, Address
f or **F**	Floating point in decimal notation	314.159265	Hex, Decimal, Binary, Boolean, ASCII, Float, Time*
e	Floating point in decimal exponent notation, lowercase `PRINTF "%e" 314.159265`	3.141593e+002	
E	Floating point in decimal exponent notation, uppercase	3.141593E+002	
g	Floating point %e or %f	314.159	
G	Floating point %E or %F	314.159	
a	Floating point in hexadecimal exponent notation, lowercase	0x1.3a28c6p+8	
A	Floating point in hexadecimal exponent notation, uppercase	0X1.3A28C6P+8	
c	Character** `PRINTF "%s%c" "Intel" 0x00AE`	Intel®	Hex, Decimal, Binary, ASCII
s	String	sample	String
p	Pointer: The offset of an address (alias for #x)	0x1000	Address

!A	Single address (TRACE32 extension), uppercase
	`PRINTF "%!A" a:100`
	`PRINTF "%!A" a:100--a:110`
		A:1000	Address, Address range

Specifier	Description	Output	Valid Argument Types
!R	Address range (TRACE32 extension), uppercase `PRINTF "%!R" a:1000--1fff`	A:1000--1FFF	Address range
n	Nothing (nothing at all)		all
* A time value is output in nanosecond in case of an integer representation, while it is output in seconds in case of a floating point representation.			
** For values from 0 to 127, the result is an ASCII character on all operating systems. For values from 128 to 255, the result depends on the font setting in the config.t32. On Windows, the result additionally depends on the active console code page.			

Flag	Description	Affected Specifier
The use of flags is optional. You may use no flag, one flag, or multiple flags.		
-	Left align: Aligns the output to the left of the minimal output width (see example). By default the output is right-aligned within the given minimal output width.	all
0	Zero padding: Left-pads numbers with zeroes instead of spaces when a minimal output width is given (see example). Not useful in combination with '-' (Left align)	d, i, o, u, x, X, f, F, a, A, e, E, g, G !y, p

#	Affix: Prepends a format-specific decoration:
	• %#x gets decoration "0x" for non-zero values
	• %#X gets decoration "0X" for non-zero values
	• %#o gets decoration "0" for non-zero values
	• %#!y gets decoration "0y" for non-zero values
	• %#!A gets decoration "0x"
	For floating point representations, a decimal point is forced. **Decimal numbers do not get a decimal point**.
	In addresses (%!A and %!R) all offsets a prefixed by "0x".
	This flag is mainly useful with addresses.
	`PRINTF "%!y" 0x42 ;result: 1000010`
	`PRINTF "%#!y" 0x42 ;result: 0y1000010`
		x, X,
		o,
		!y,
		f, F, g, G
		!A, !R

+	Force algebraic sign: • Positive numbers are prefixed by a plus sign (+). • Negative numbers are prefixed by a minus sign (-).	d, i, f, F, a, A, e, E, g, G
(space)	Positive numbers are prefixed by a space character. Negative numbers are prefixed by a minus sign (-).	d, i, f, F, a, A, e, E, g, G

Width	Description
The optional width specifies the minimum output width and is either a decimal number or an asterisk:	
<number>	The minimum number of characters to be written including an optional algebraic sign or prefix/suffix (with flag **#**). • If the representation of the data uses fewer characters, the output is usually padded with blank spaces. • If flag **0** is used, the output is padded with zeroes instead. • If the representation of the data requires more characters, it is not truncated. `PRINTF "%08x" 0x42`
*	By using an asterisk inside the *<format_string>*, the value for the minimum output width is taken from an additional *<data>* argument. This additional argument precedes the argument that has to be formatted (and also precedes any possible argument for the precision). `PRINTF "%0*x" 8 0x42`

Precision	Specifier	Description
The meaning of the optional precision depends on the specifier at the end of the control sequence. It is either a number or an asterisk:		
.*<number>*	d, i, u, o, x, X, !y, p	The minimum number of digits to be written. If the written number has fewer digits than specified with the precision, the remaining digits are padded with leading zeroes.
	f, F, a, A, e, E	The number of digits after the decimal point.
	g, G	The maximum number of significant digits to be written.
	s	The maximum number of characters to be written.
	c, n	No effect. Precision value is ignored.
	!A, !R	The minimum number of digits for the address offset. With addresses, you may also specify one or two additional precision values - each starting with a decimal point: • %[flags][width][.precision[.precision[.precision]]][length]specifier See example. The precision values specify the minimum number of digits for the memory segment and/or machine ID, if the address contains a memory segment and/or machine ID.
.*	By using an asterisk inside the *<format_string>*, the value for the precision is taken from an additional *<data>* argument. This additional argument precedes the argument that has to be formatted. `PRINTF "%*.*x" 8 4 0x42`	

Length	Description	Affected Specifier
The optional length is mainly accepted for compatibility to format strings used with printf() in C/C++. However, the length has a slightly different meaning in TRACE32 than with C/C++. In TRACE32 the length can specify a bit limit for integer representations. The default length for all integer values is 64-bit.		
h	Truncate the output value to 16-bit.	d, i, u, o, x, X, !y
hh	Truncate the output value to 8-bit.	d, i, u, o, x, X, !y
l	Truncate the output value to 32-bit.	d, i, u, o, x, X, !y
ll	Truncate the output value to 64-bit (default).	d, i, u, o, x, X, !y
L, **j**, **z** or **t**	Silently ignored length specifiers (for compatibility to format strings used with printf() in C/C++)	all

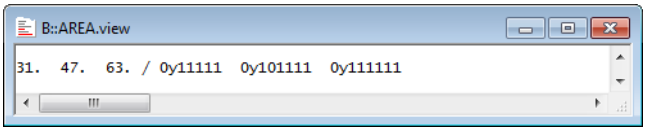
Example 1

```
;print hex values in decimal and in binary format
;       <format_string>                    <data>
PRINTF "%i.  %i.  %i. / %#!y  %#!y  %#!y" 0x1f 0x2f 0x3f 0x1f 0x2f 0x3f
```

% = Control character **i** = Specifier **#** = Flag **!y** = Specifier

. = Postfix for decimal values, see **SETUP.RADIX**.

Output:



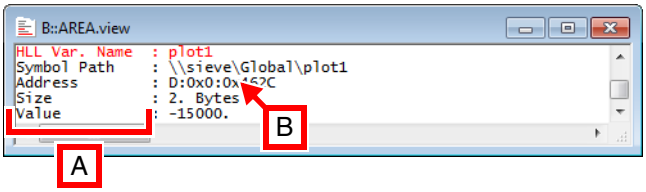
Example 2

```
PRIVATE &hll_variable ;declare a PRACTICE macro
Data.Load.ELF "~~/demo/arm/compiler/gnu/sieve.elf" /RelPATH

;assign name of HLL variable from *.elf file to the PRACTICE macro
&hll_variable="plot1"

;strings padded to a column width of 15 characters
PRINTF %COLOR.RED "%-15s: %s" "HLL Var. Name" "&hll_variable"
PRINTF "%-15s: %s"            "Symbol Path"   sYmbol.NAME(&hll_variable)
PRINTF "%-15s: %#!A"          "Address"       Var.ADDRESS(&hll_variable)
PRINTF "%-15s: %i. Bytes"     "Size"          Var.SIZEOF(&hll_variable)
PRINTF "%-15s: %+i."          "Value"         Var.Value(&hll_variable)
```

% = Control character **-** = Flag **15** = Width **s** = Specifier; for result, see [**A**] below.

% = Control character **#** = Flag **!A** = Specifier; for result, see [**B**] below.

Output:



A Left-aligned column with a width of 15 characters (**%-15s**)

B Single address in upper case with the prefix 0x (**%#!A**)

Example 3

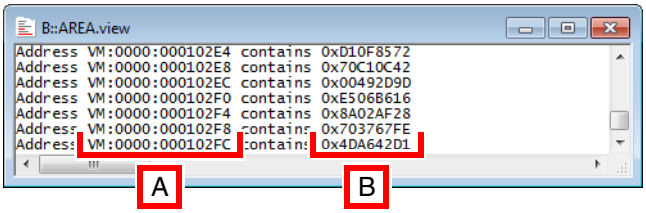
Random data and space IDs from the TRACE32 virtual memory (VM:) are printed in fixed columns.

```
PRIVATE &i &addr
&i=0
&addr=VM:0x10200

SYStem.Option.MMUSPACES ON        ;enable space IDs for logical addresses

Data.PATTERN &addr++0xFF /RANDOM ;create a random pattern

WHILE &i<0x100
(
  PRINTF "Address %.8.4!A contains 0x%08X" &addr+&i Data.LONG(&addr+&i)
  &i=&i+4
)
```

% = Control character **.8.4** = Precision **!A** = Specifier; for result, see [**A**] below.

% = Control character **0** = Flag **8** = Width **X** = Specifier; for result, see [**B**] below.

Output:



A Prints an address with an offset of at least 8 characters and its space ID of at least 4 characters (**%.8.4!A**)

B Prints data in hexadecimal form with at least 8 characters. The data is left-padded with zeros if the data has less than 8 characters (**%08X**).

See also

■ PRINT ■ ECHO ■ SPRINTF

▲ 'Release Information' in 'Legacy Release History'

[Example]

Format:	**PRIVATE** {*<macro>*}

Creates a private macro. PRACTICE macros declared with **PRIVATE** exist inside the declaring block and are erased when the block ends. They are only visible in:

Yes The declaring block and all sub-blocks (e.g. **IF**..., **RePeaT**..., **WHILE**..., etc.)

No Subroutines (**GOSUB**...**RETURN**)

No Sub-scripts (**DO**...**ENDDO**)

Example: The following script is just intended to illustrate the usage of private macros. To try this demo script, simply copy the script to a `test.cmm` file, and then step through the script (See "**How to...**").

```
LOCAL &msg          //declare LOCAL macro
&msg="Hello World!"

PRINT "&msg"        //show LOCAL macro
GOSUB child1        //call a subroutine
PRINT "&msg"        //show LOCAL macro
ENDDO
;-----------------------------------------------------------------------
child1:
    PRIVATE &msg    //declare PRIVATE macro
    &msg="This comes from a private macro"

    //show PRIVATE macro defined in child1
    PRINT "child1 says: &msg"

    (
        //PRIVATE macros are visible inside a sub-block:
        &msg="This comes again from the private macro"
        PRINT "&msg"
    )
    GOSUB grandchild1

    //show PRIVATE macro defined in child1
    PRINT "child1 says: &msg"
    RETURN
;-----------------------------------------------------------------------
grandchild1:
    //LOCAL macros visible in any subroutine:
    //show again LOCAL macro defined in the beginning of the script
    PRINT "grandchild1 says: &msg"
    &msg="Hello Universe!"
    RETURN
```

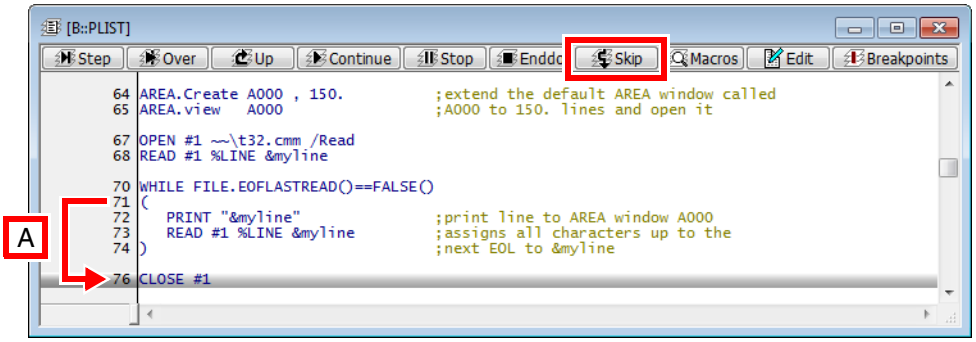
See also

- PMACRO.EXPLICIT
- PMACRO.IMPLICIT
- PMACRO.list
- ENTRY
- GLOBAL
- LOCAL
- SPRINTF

▲ 'Release Information' in 'Legacy Release History'

Format:	**PSKIP**

Skips the current command or block in a PRACTICE script without executing command or block. The PC moves to the next executable PRACTICE script line, but does not execute it.



A Clicking **Skip** in line 71 skips the block without executing it and moves the PC to line 76.

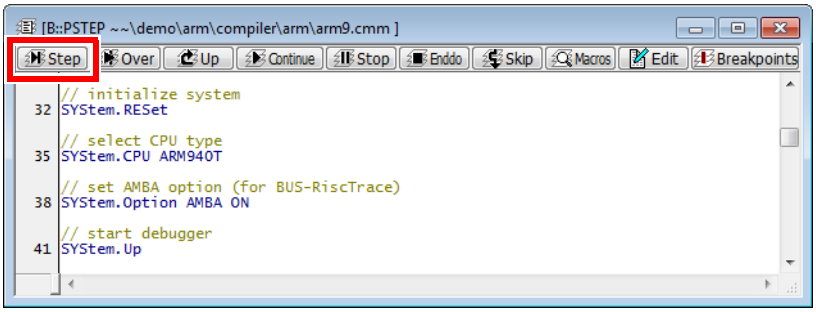
As an alternative to **PSKIP**, you can right-click the command where you want to continue the PRACTICE script execution, and then select **Set PC here** from the popup menu.

What is the difference between ...?

PSKIP	**Set PC here**
Automatically moves the PC to the next executable line.	Lets *you* place the PC to an executable line where *you* want to continue.

Format:	**PSTEP** [*<file>* [*<parameter_list>*]]

If no PRACTICE script is loaded, the command will toggle the run mode. If the run mode is switched to single step, the next started PRACTICE script will stop at the first line and can be executed in single step mode in the **PSTEP** window.




```
PSTEP                          ; sets PRACTICE script execution to
                               ; single step mode
DO test                        ; starts PRACTICE script
PSTEP                          ; single step in PRACTICE script
```

With a given PRACTICE script file name the **DO** command will be superfluous.

```
PSTEP test.cmm 0x1fff          ; sets PRACTICE script execution to
                               ; single step mode and starts the
                               ; PRACTICE script
```

See also

- PSTEPOUT
- PSTEPOVER
- PBREAK
- PEDIT
- PLIST
- DO
- RUN

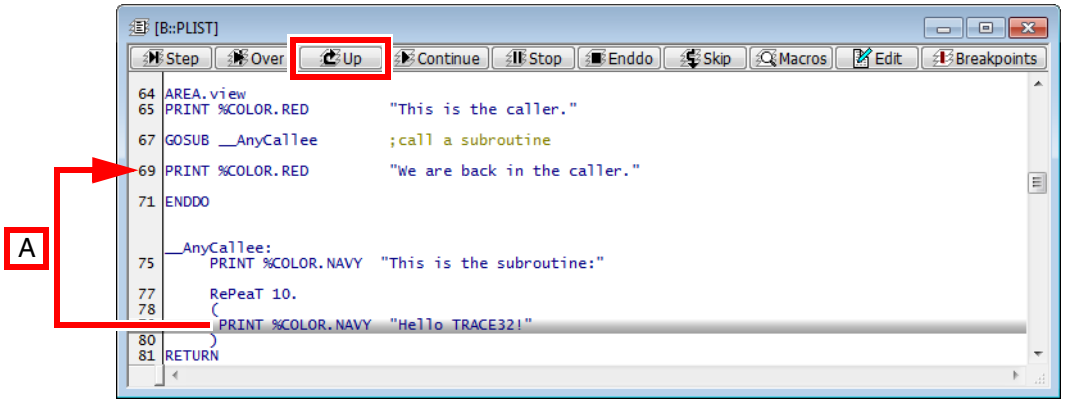
▲ 'Release Information' in 'Legacy Release History'
▲ 'Debugging of PRACTICE Script' in 'Training Script Language PRACTICE'

Format:	**PSTEPOUT**

Executes all PRACTICE script lines of a callee, returns to the caller, and stops PRACTICE script execution at the next executable script line of the caller. A callee can be a PRACTICE subroutine (**GOSUB**...**RETURN**) or a sub-script (**DO**...**ENDDO**).

Prerequisite: You have run the **PSTEP** command to switch to the PRACTICE script single-step mode.

Clicking the Up **Up** button in the PRACTICE script windows **PLIST**, **PSTEP**, or **PMACRO.list** executes the **PSTEPOUT** command via the user interface:



A **PSTEPOUT** runs the subroutine to completion, returns to the caller and stops PRACTICE script execution at the next executable script line of the caller.

See also

- PSTEPOVER
- PSTEP

▲ 'Release Information' in 'Legacy Release History'

Format:	**PSTEPOVER**

Executes a PRACTICE script line and stops script execution at the next executable script line. A callee such as a PRACTICE subroutine (**GOSUB**...**RETURN**) or sub-script (**DO**...**ENDDO**) is run to completion *without leaving the caller*.

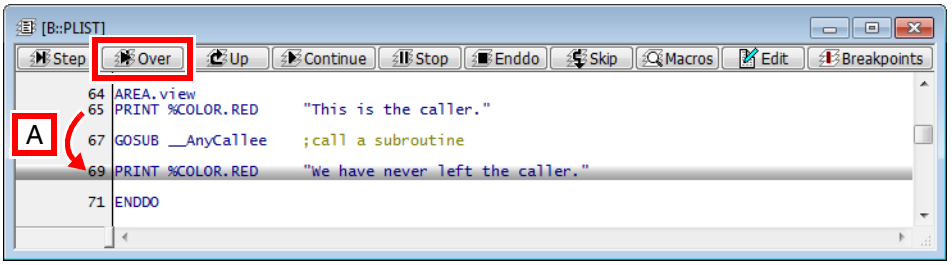
Prerequisite: You have run the **PSTEP** command to switch to the single-step mode for PRACTICE scripts.

NOTE: If there is an active breakpoint in the callee you are trying to step over, then script execution will stop at the breakpoint (see also **PBREAK**).

A **PSTEPOVER** will also be stopped by the following commands in a callee:

- **ENTER**
- **INKEY**
- **STOP**

Clicking the Over **Over** button in the PRACTICE script windows **PLIST**, **PSTEP**, or **PMACRO.list** executes the **PSTEPOVER** command via the user interface:



A **PSTEPOVER** runs the subroutine to completion *without leaving the caller*, and stops PRACTICE script execution at the next executable script line of the caller.

See also

- PSTEPOUT
- PSTEP

READ — Read from data file

Format: **READ #**<*buffer_number*> [**%LINE**] <*parameter_list*>

Read data from an open file. Arguments in the file are separated by blanks.

LINE With the **LINE** option a complete line is read into a PRACTICE macro.

Examples:

```
        LOCAL &offset &data &headerline
        OPEN #1 datafile.dat /Read

        READ #1 %LINE &headerline
        PRINT "&headerline"

        // script for newer software versions

        RePeaT 10.
        (
           READ #1 &offset &data
           IF EOF()
                GOTO endloop
           Data.Set &offset &data
        )
  endloop: CLOSE #1
```

```
        // for older software versions than 2008.03.03
        RePeaT 10.
        (
            READ #1 &offset &data
            IF "&offset"==""
                GOTO endloop
            Data.Set &offset &data
        )
  endloop: CLOSE #1
```

See also

- CLOSE
- OPEN
- WRITE

Format 1:	**RePeaT** *<count>* *<command>*
Format 2:	**RePeaT** [*<count>*] *<block>*
Format 3:	**REPEAT** *<block>* **WHILE** *<condition>*

The command or script block following a **RePeaT** command will be executed *<count>* times (format 1 and 2) or once and then as long as the *<condition>* is true (format 3).

If *<count>* is set to 0, the loop is executed as an endless loop.

Example of format 1: The execution of a single command is repeated 10 times. The **RePeaT** command may be started interactively from the command line.

```
AREA.view
RePeaT 10. PRINT "X"
```

Example of format 2:

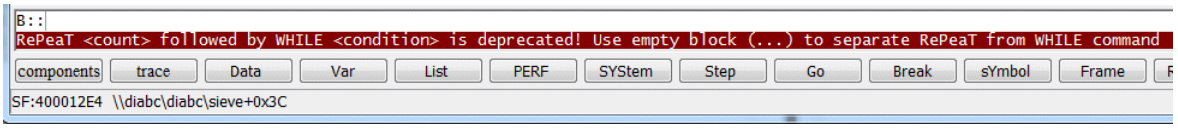
```
…
Var.Break.Set flags /Write            ; set a Write breakpoint to array
                                      ; flags
; repeat the following 10. times
; start program and wait until program execution is stopped at breakpoint
; export contents of array flags to file flags_export.csv in CSV format
REPEAT 10.
(
  Go
  WAIT !STATE.RUN()
  Var.EXPORT "flags_export.csv" flags /Append
)
…
```

Example of format 3:

```
; read line from file my_strings.txt
; write not-empty lines to file my_strings_noempty.txt

PRIVATE &CurrentLine
OPEN #1 my_strings.txt /Read
OPEN #2 my_strings_noempty.txt /Create

RePeaT
(
    READ #1 %LINE  &CurrentLine
    IF (!FILE.EOFLASTREAD()&&("&CurrentLine"!=""))
       WRITE #2 "&CurrentLine"
)
WHILE !FILE.EOFLASTREAD()
CLOSE #1
CLOSE #2
```

The following command sequence will generate an error message when the script is started:

```
; below command sequence is illegal and throws an error!
RePeaT <count>
  <block>
WHILE <condition>
   <block>
```



The following workaround solves this problem:

```
RePeaT <count>
  <block>
(
  ; empty block to separate RePeaT command from WHILE command
)
WHILE <condition>
   <block>
```

See also

■ DO ■ WHILE

▲ 'Release Information' in 'Legacy Release History'

Format:	**RETURN** [*<parameter_list>*]

The subroutine is finished. Optionally, parameters may be passed, which can be taken over using the **ENTRY** command.

Example:

```
GOSUB subr2
ENTRY &portval
…

subr2:
     &result=Data.Byte(sd:0x100)
     RETURN &result
```

See also

- ENTRY
- GOSUB

Format:	**RETURNVALUES** {*<macro>*}

Takes the return values of a PRACTICE script/subroutine. The return values have to be enclosed in quotes (") in the call. An error message is generated, if a macro name is used, that cannot be found in the current scope.

Examples:

```
PRIVATE &sr1_dec &sr1_hex &sr1_string

GOSUB sr_1
RETURNVALUES &sr1_dec &sr1_hex &sr1_string
ENDDO

sr_1:
(
  RETURN "5." "0x55" "Okay"
)
```

```
PRIVATE &range &boolean &symbol &val1 &val2 &strA

GOSUB AnySubroutine

;take the return values passed by the subroutine
RETURNVALUES &range &boolean &symbol &val1 &val2 &strA

PRINT "&range &boolean &symbol &val1 &val2 &strA"
ENDDO


AnySubroutine:
PRIVATE &my_rng &my_bool &my_symb &my_val1 &my_val2 &my_strA

    &my_rng="0x40000000++0xffff"    ;any range
    &my_bool=FOUND()                ;any boolean expression
    &my_symb="\MCC\sieve"           ;any symbol
    &my_val1="10."                  ;any decimal value
    &my_val2="0xA"                  ;any hex value
    &my_strA="Hello TRACE32!"       ;any string

RETURN "&my_rng" "&my_bool" "&my_symb" "&my_val1" "&my_val2" "&my_strA"
```

See also

- ENTRY
- PARAMETERS

▲ 'Release Information' in 'Legacy Release History'

Format:	**RUN** *<file>* [*<parameter_list>*]

Starts a PRACTICE script after clearing the old PRACTICE stack. Additional parameters may be defined which are passed to the subroutine. The command is identical to the **DO** command, except that it clears the old stack before starting the PRACTICE module.

See also

- DO
- ENDDO
- PSTEP

SCREEN — Screen updating

If PRACTICE scripts are executed, the screen is normally not updated. The **SCREEN** commands allow updating of the screen manually or automatically. The command has **no effect** on the screen update, while PRACTICE is not running, e.g. the update at spot points.

See also

- SCREEN.ALways
- SCREEN.display
- SCREEN.OFF
- SCREEN.ON
- SCREEN.WAIT

SCREEN.ALways — Refresh always

Format:	**SCREEN.ALways**

Refreshes the screen after every PRACTICE line. This slows down the execution speed, but allows tracing the script flow.

Example:

```
PLIST                          ; display script state
SCREEN.ALways
DO test.cmm                    ; run
```

See also

- SCREEN

SCREEN.display — Refresh screen

Format: **SCREEN.display**

Updates the screen.

Example:

```
SCREEN.OFF                      ; turn refresh off for fast printing
PRINT " Screen Mask:   A"
PRINT "                B"
PRINT …
…
SCREEN                          ; update the screen
```

See also

- SCREEN

SCREEN.OFF — No refresh

Format: **SCREEN.OFF**

No refreshing of the screen is done, while PRACTICE scripts are running (except **SCREEN.display** commands). This allows PRACTICE scripts to execute very fast, but the results will not be seen on the screen during script execution.

See also

- SCREEN

SCREEN.ON — Refresh when printing

Format: **SCREEN.ON**

Refreshes the screen after every **PRINT** command.

See also

- SCREEN

Format:	**SCREEN.WAIT** [*<condition>* | *<period>*]

Same as **WAIT**, but updates the screen while waiting. If you use a terminal window or if you display variables with **run-time memory access**, this might be required.

If the command **SCREEN.WAIT** is used without parameters, the PRACTICE script waits until all processing windows are completed before the next PRACTICE instruction is interpreted. Examples of processing windows are: **Trace.STATistic.Func**, **Trace.Chart.sYmbol** or **CTS.List**.

<condition>	PRACTICE functions that return the boolean values TRUE or FALSE.
<period>	Min.: 1ms Max.: 100000s Without unit of measurement, the specified value will be interpreted as seconds and must be an integer. See below.

Example 1:

```
…                                       ; configure TERMinal window

TERM.view                               ; display TERMinal window

Go                                      ; start program

SCREEN.WAIT 5.s                         ; wait 5.s and update screen while
                                        ; waiting

…
```

Example 2:

```
…

Go                                      ; start program execution

WAIT !STATE.RUN()                       ; wait until program execution
                                        ; stopped

Trace.Chart.sYmbol                      ; perform a flat function run-time
                                        ; analysis

SCREEN.WAIT                             ; update screen and wait until
                                        ; flat function run-time analysis
                                        ; is completed

…                                       ; continue PRACTICE script
                                        ; execution
```

See also

- SCREEN
- PRinTer.OPEN
- TIMEOUT
- WAIT

▲ 'Release Information' in 'Legacy Release History'

Format:	**SPRINTF** *<macro>* "*<format_string>*" [{*<data>*}]

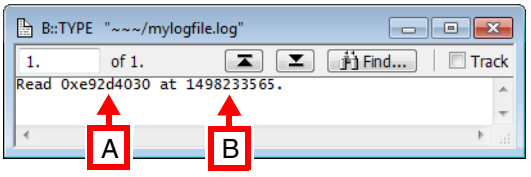
Writes text and formatted data to a PRACTICE macro in the style of the sprintf() function of C/C++.

<macro>	If a PRACTICE macro with the given name does not exist, then it is created on the local PRACTICE stack.
<format_string>	The characters of the *<format_string>* are written to the PRACTICE *<macro>*.
%	Each percent sign (**%**) inside the *<format_string>* starts a control sequence which is replaced by the *<data>* arguments following the *<format_string>*. The control sequence started with the percent sign has the form: • **%**[flags][width][.precision][length]specifier For details on the control sequence and other special characters inside the *<format_string>*, see the **PRINTF** command.

Example:

```
PRIVATE &str                ;declare a PRACTICE macro of the type PRIVATE

SPRINTF &str "Read 0x%08x at %10i." Data.Long(A:0x100) DATE.UnixTime()

APPEND "~~~/mylogfile.log" "&str"

TYPE   "~~~/mylogfile.log" ;display the result in the TYPE window
```

% = Control character **0** = Flag **8** = Width **x** = Specifier; for result, see [**A**] below.
10 = Width **i** = Specifier . = Postfix for decimal values (**SETUP.RADIX**); for result, see [**B**] below.



See also

■ ECHO ■ GLOBAL ■ LOCAL ■ PRINT
■ PRINTF ■ PRIVATE

▲ 'Release Information' in 'Legacy Release History'

Format:	**STOP** [*<parameter_list>* ...]

The script is stopped, but it remains in the working storage and can be reactivated by the **CONTinue** command. The arguments are displayed in the message line and **AREA** window (same as command **PRINT**).

Example:

```
Data.Test 0x0--0x0ffff
IF FOUND()
     STOP  "Memory error"
```

See also

- CONTinue
- END
- ENDDO
- WAIT

Format:	**SUBROUTINE** *<name>*

Defines a subroutine in a PRACTICE script. A subroutine must be defined at the top level of the script, it is not allowed to define subroutines within a block. The SUBROUTINE statement must be followed by a block.

A subroutine block should end with a **RETURN** statement. If no return values are required, the RETURN statement is optional. If the block end is reached without reaching a RETURN statement, the subroutine will return to the caller without passing return values (implicit return).

Call subroutines using **GOSUB**. Example:

```
PRINT "Performing memory test..."

SUBROUTINE BoardSetup
(
  SYStem.CPU CortexA5
  SYStem.Up
  ;implicit RETURN
)

SUBROUTINE MemoryTest
(
  PRIVATE &address &mem_ok
  PARAMETERS &address
  Data.Set &address %Long 0x5A5A5A5A
  &mem_ok=Data.LONG(&address)==0x5A5A5A5A
  RETURN "&mem_ok" ;explicit RETURN
)

GOSUB BoardSetup
PRIVATE &test1 &test2
GOSUB MemoryTest "ANC:0x1000"
RETURNVALUES &test1
GOSUB MemoryTest "ANC:0x2000"
RETURNVALUES &test2

IF (&test1)&&(&test2)
  PRINT "Memory OK."
```

NOTE:
- GOSUB accepts both labels and subroutine names as target, therefore labels and subroutines can not have the same name.

WAIT — Wait until a condition is true or a period has elapsed

Format:	**WAIT** [*<condition>*] [*<period>*] [/**RunTime**]

Waits for the specified condition to become true or for the specified period to elapse. If both a condition and a period are specified, then the first argument to enter the desired state terminates the command. The granularity of period, as well as the minimum period time is 1ms. The maximum period is 100000s.

While waiting the screen is not updated. If you want the screen to be updated while waiting use the command **SCREEN.WAIT**.

<condition>	PRACTICE functions that return the boolean values TRUE or FALSE.
<period>	Min.: 1ms Max.: 100000s Without unit of measurement, the specified value will be interpreted as seconds and must be an integer. See below.
RunTime	Wait period depends on the target runtime and not on the host time.

Example 1: Run target program for 1 second.

```
Go
WAIT 1s
Break
```

Example 2: Wait until core halts at a breakpoint.

```
Break.Set sieve
Go
WAIT !STATE.RUN()
```

Example 3a: Wait until core halts at a breakpoint, with 2s timeout.

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

IF STATE.RUN()
(
  PRINT %ERROR "function main not reached!"
  ENDDO
)
```

Example 3b: Wait until core halts at a breakpoint, with 2s timeout.

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

IF TIMEOUT()
(
  PRINT %ERROR "function main not reached!"
  ENDDO
)
```

See also

- WHILE
- INKEY
- SCREEN.WAIT
- STOP
- TIMEOUT
- SYStem.PAUSE
- TIMEOUT()

Format:	**WHILE** [*<condition>*]

The command or script block following a **WHILE** statement will be executed as long as the condition is true. Emulators and debuggers have an counterpart of this command that works in the HLL syntax of the target program (command **Var.WHILE**).

Examples:

```
WHILE Register(d0)==0x0
(
     Register.Set pc testprog
     Go testend
)
```

```
Var.WHILE flags[9]!=0       ; HLL expression in condition
    Step
```

NOTE:	**`WHILE`** must be followed by a white space.

See also

- WAIT
- RePeaT
- Var.WHILE
- FILE.EOF()
- FILE.EOFLASTREAD()
- STATE.RUN()

▲ 'Release Information' in 'Legacy Release History'

Format:	**WRITE #**<*buffer_number*> [**%**<*format*>] <*parameter_list*>
<*format*>:	**CONTinue**

Save data in a data file. The syntax of the command is similar to the **PRINT** command.

CONTinue	Continues to print data to the current line (and not to a new line).

Example: To test this script, simply copy it to a `test.cmm` file, and then run it in TRACE32 (See "**How to...**").

```
;create a file in the temporary directory of TRACE32 and
OPEN #1 ~~~/datafile.dat /Create       ;open the file for writing

&offset=0x2228

WHILE &offset<0x22C4
(                                      ;write data to file
   WRITE #1 "At Address " &offset " is Data " Data.Long(SR:&offset)
   &offset=&offset+1
)

CLOSE #1                               ;close the file for writing

TYPE ~~~/datafile.dat /LineNumbers     ;optional: open file in TYPE win.
```

The path prefix ~~~ expands to the temporary directory of TRACE32.

See also

- WRITEB
- APPEND
- CLOSE
- MKTEMP
- OPEN
- READ

▲ 'Release Information' in 'Legacy Release History'

Format:	**WRITEB #**<*buffer_number*> [**%**<*format*>] <*data*> | <*string*>
<*format*>:	**Byte** | **Word** | **Long** | **BE** | **LE**

Writes binary data to a file. The option **/Binary** for the **OPEN** command allows to open or create binary files.

Example:

```
;create a binary file in the temporary directory of TRACE32 and
;open the binary file for writing
OPEN #1 ~~~/test.bin /CREATE /BINARY

WRITEB #1 Var.VALUE(flags[0])     ;write the values of an HLL variable
WRITEB #1 Var.VALUE(flags[1])     ;to the binary file
WRITEB #1 Var.VALUE(flags[3])
WRITEB #1 Var.VALUE(flags[4])

CLOSE #1                          ;close the binary file for writing

DUMP ~~~/test.bin                 ;display a binary file in hex
                                  ;and ASCII format.
```

See also

- WRITE
- CLOSE
- OPEN

▲ 'Release Information' in 'Legacy Release History'