LAUTERBACH
DEVELOPMENT TOOLS

# PRACTICE Script Language User's Guide

Release 09.2024

# PRACTICE Script Language User's Guide

# PRACTICE Script Language User’s Guide

**Version 05-Oct-2024**

## History

02-Nov-2022   In the chapter ‘Related Documents‘ a reference to ide_user.pdf has been added.

## Why Use PRACTICE Scripts

Using PRACTICE scripts (*.cmm) in TRACE32 will help you to:

- Execute commands right on start of the debugger
- Customize the TRACE32 PowerView user interface to your project requirements
- Set up the debugger with settings for the target board
- Standardize repetitive and complex actions
- Initialize the target (e.g. the memory to load an application)
- Load the application and / or the symbols
- Add your own and extend available features
- Speed up debugging through automation
- Share the debugger methods with other users and empower them to work more efficiently
- Make debug actions reproducible for verification purposes and regression tests

## 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.
- **“Training Script Language PRACTICE”** (training_practice.pdf): Describes how to run and create PRACTICE script files (*.cmm).
- **“PRACTICE Script Language Reference Guide”** (practice_ref.pdf)
- **PRACTICE Reference Card** (support.lauterbach.com/downloads/files/practice-reference-card-pdf-2)
- **Video Tutorials** (support.lauterbach.com/kb/articles/practice-tutorial)

# PRACTICE Script Structure

## Function

PRACTICE is a line-oriented test language which can be used to solve all usual problems of digital measurement engineering. PRACTICE-II is an enhanced version of this test language, first developed in 1984 for in-circuit emulators.

The test language allows interactive script development with the possibility of a quick error removal and an immediate script execution. The execution of PRACTICE testing scripts can be stopped and restarted at any time.

PRACTICE contains an extremely powerful concept for handling script variables and command parameters. This macro concept allows to substitute parameters at any point within the commands. Since PRACTICE variables can only occur as **PRACTICE macros**, conflicts between target program names are ruled out.

## Difference between Variables and PRACTICE Macros

PRACTICE macros are based on a simple text replacement mechanism similar to C preprocessor macros. However, in contrast to C preprocessor macros, a PRACTICE macro can change its contents during it's life time by simply assigning a new value.

Each time the PRACTICE interpreter encounters a macro it is replaced with the corresponding character sequence. Only after performing all text replacements the resulting line is interpreted (just like the C compiler only works on the fully preprocessed text).

PRACTICE macros are declared using the commands **PRIVATE** or **LOCAL** or **GLOBAL**.

The *visibility* of PRACTICE macros differs notably from other script languages (unless declared using the **PRIVATE** command): they are accessible from all subsequently executed code while they are alive e.g. in:

- Subroutines (**GOSUB** … **RETURN**)
- Sub-scripts (**DO** … **ENDDO**)
- Sub-blocks (**IF** …, **RePeaT**, **WHILE**, etc.)

| | |
|---|---|
| **NOTE:** | PRACTICE does not know the concept of variables with a corresponding type like uint32 or uint8 in C. |

# PRACTICE Script Elements

PRACTICE scripts consist of labels, commands, and comments:

| | |
|---|---|
| `;example` | comment starting with `;` |
| `//example` | comment starting with `//` |
| `start:` | label |
| `Step` | command |
| `GOTO start` | command and label |
| `B::` | command to change the default device |
| `B::Data.dump` | command preceded by a device selector |

## Labels

Labels always start in the first column and are always followed by a colon. Labels are case sensitive.

## Comments

Comments are prefaced by a semicolon `;` or two forward slashes `//`.
Inline comments for **Var.*** commands *must start* with two forward slashes `//`.

```
Var.set func7(1.5,2.5) //execute a function in the target
```

## Line Continuation Character

To continue a string on the next line, a backslash `\` is used to indicate a line continuation in a PRACTICE script (*.cmm). No white space permitted after the backslash. If the line continuation character is used at the end of a comment line, then the next line is interpreted as a comment line as well.

```
DIALOG.OK "Please switch ON the TRACE32 debugger first"+\
" and then switch ON the target board."
```

## Script Flow

Several commands allow to control the script flow. Scripts may be divided in several modules. Subroutines within a module are called by the **GOSUB** command, another module is called by the **DO** command.

| | |
|---|---|
| **STOP** | Stop temporarily |
| **END** | Terminate script and clear stack |
| **CONTinue** | Continue with script execution |
| **DO** | Call script module |
| **ENDDO** | Terminate script module |
| **RUN** | Clear PRACTICE stack and call module |
| **GOSUB** | Call subroutine |
| **RETURN** | Return from subroutine |
| **GOTO** | Branch within module |
| **JUMPTO** | Branch to other module |

## Conditional Script Flow

Conditional script execution is made by several commands:

| | |
|---|---|
| **IF** | Conditional block execution |
| **ELSE** | Block that is only compiled when the IF condition is false |
| **WHILE** | Conditional script loop |
| **RePeaT** | Repetitive script loop |
| **ON** | Event-controlled PRACTICE script execution |
| **GLOBALON** | Global event-controlled PRACTICE script execution |
| **WAIT** | Wait for event or delay time |

```
IF    OS.FILE(data.tst)
      PRINT "File exists"
ELSE
      PRINT "File doesn't exist"

WHILE Register(pc)==0x1000             ; step until pc = 1000H
      Step

RePeaT 100. Step                       ; step 100 times

RePeaT   0. Step                       ; step endless

ON ERROR GOTO errorexit
```

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

## Script Nesting

PRACTICE scripts can be nested hierarchically. A second script may be called as a subroutine from the initial script. This subroutine on its part may call up a third script as a subroutine. This allows a structured, modular script development.

```
; Script containing two script calls

PRINT "Start"
DO modul1                         ; Execute script module 1
DO modul2                         ; Execute script module 2
                                  ; the file extension (*.cmm) can be
ENDDO                             ; omitted
```

## Block Structures

Several PRACTICE commands can be combined to form a block. A block is a collection of commands, which are always executed simultaneously. Blocks usually require **IF**, **WHILE**, or **RePeaT** statements. They can, however, be implemented anywhere in order to mark connective blocks. Blocks are marked by round parentheses.

You can jump out of a block, but not into a block.

```
; Block nesting
start:
     IF &abc
     (
          PRINT "Function 1"
          DO func1
     )
     ELSE
     (
          PRINT "Function 2"
          DO func2
          IF &xyz GOTO start  ;jump out of the block to the label ‘start’
     )
ENDDO
```

## PRACTICE Macros

PRACTICE macros are **character sequences with a maximum size of 4KB**. The character sequence is interpreted in the context where it is used. For a command, a PRACTICE macro can be interpreted e.g. as a number, boolean expression, parameter, or simply as a string. PRACTICE macros can be expanded to complete commands, too.

PRACTICE macros are generated by an allocation. PRACTICE macros found in a script file line are replaced by the text they contain (except special commands, e.g. **ENTRY**). The macros can be put into parentheses, if necessary. The double ampersand form ('&&') of the assignment forces a recursive macro resolution if the resolved value still contains macro names. Macros may be defined and modified inside the interactive command line.

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

**Macro expansion does NOT take place inside the TRACE32 command line!**

```
e.g.      PRINT     &macroname                      or

          Data.List &my_startaddress
```

**Macro expansion only works in PRACTICE scripts (*.cmm):**

| | |
|---|---|
| normal: | **&***<macroname>***=***<expression>* |
| recursive: | **&&***<macroname>***=***<expression>* |

```
&int=1
&int=&int+1                           ; increment current value of &int
&text="This is a test"
&command="Data.dump Register(PC)"
&float=1.4e13
&range="0x1000--0x1fff"
&address=func5
&addressrange=P:0x1234..0x5555
&boolean=SYStem.Up()

PRINT &int
PRINT "&int"                          ; after replacement: 0x2
PRINT "&(int)nd"                      ; after replacement: "2nd"
ENDDO
```

PRACTICE macros can be declared as **GLOBAL** or **LOCAL** or **PRIVATE** macros.

| | |
|---|---|
| **LOCAL** | Declare local macros |
| **GLOBAL** | Declare global macros |
| **PRIVATE** | Declare private macros |

## GLOBAL Macros

PRACTICE macros declared with **GLOBAL** are accessible at every script level and have an unlimited life-time.

## LOCAL Macros

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.) |

## PRIVATE Macros

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**) |

| | |
|---|---|
| **NOTE:** | If a value is assigned to a macro which does not yet exist on the PRACTICE stack or is an inaccessible **PRIVATE** macro, then the macro is implicitly created as a **LOCAL** macro. |

## Switching PRACTICE Macro Expansion ON or OFF

You can switch the PRACTICE macro expansion ON or OFF in the following *embedded* script blocks:

- **DIALOG.view** blocks *embedded* in PRACTICE script files (*.cmm), as shown in the example below.
- **MENU.ReProgram** blocks *embedded* in PRACTICE script files (*.cmm)
- PRACTICE script block *embedded* in dialog files (*.dlg)
- PRACTICE script block *embedded* in menu files (*.men)

An embedded block is delimited by opening round brackets "`(`", "`(&`", "`(&+`" or "`(&-`" and closed by closing round brackets "`)`".

| | |
|---|---|
| **ON** | Use `(&` or `(&+` as opening block delimiter to switch ON macro expansion for this block and its sub-blocks. |
| **OFF** | Use `(&-` as opening block delimiter to switch OFF macro expansion for this block and its sub-blocks. |

Switching macro expansion ON is useful, for example, if you want to make a button text configurable.

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

```
LOCAL &btn

&btn="BUTTON ""Macro is expanded when script is loaded"" \
""PRINT """"It works!"""""""

DIALOG.view
(&+ ;macro expansion of the macro &btn needs to be
    ;switched on in this DIALOG.view block, which is embedded in
    ;a PRACTICE script
    POS 0. 0. 35. 1.
    &btn

    BUTTON "Hardcoded button text"
    (&- ;in this sub-block, macro expansion is switched off
        &btn="No expansion here"
        PRINT "&btn"
    )
)
STOP
```

## Parameter Passing

Parameters can be passed to a subroutine with a parameter list. The parameters are assigned to local PRACTICE macros, using the **ENTRY** command within the subroutine.

Subroutines can also be called up interactively by parameters. Arguments can be passed by the commands **DO**, **ENDDO**, **RUN**, **GOSUB**, **RETURN**. They can also be the result of calling a command extension or invoking the driver program on the host with arguments.

**White spaces before or after operators inside expressions are interpreted as separators of consecutive parameters.**

| **ENTRY** | Parameter passing |
|---|---|

```
SYStem.Up

GOSUB myTEST 0x0--0xfff

ENDDO

;by calling the subroutine myTEST, a memory test of the address range
;0x0--0xfff can be executed
myTEST:
   ENTRY     &range
   Data.Test &range
RETURN
```

## Input and Output

Several input and output commands allow interaction with the user. Input is normally done through an **AREA** window. All print operations are displayed on the TRACE32 message line. The **AREA** window **A000** is displayed by default. Inputs and outputs can be re-routed to another **AREA** window.

| | |
|---|---|
| **PRINT** | Print to screen |
| **BEEP** | Stimulate sound generator |
| **ENTER** | Window based input |
| **INKEY** | Character input |

```
PRINT "The PC address is " Register(pc)    ; print message

BEEP                                       ; end acoustic signal

INKEY                                      ; wait for keystroke

INKEY &char                                ; wait for keystroke
IF &char=='A'
     GOSUB func_a
IF &char=='B'
      GOSUB func_b
…

AREA.Create IO-Area                        ; create a window named
                                           ; IO-Area
AREA.Select IO-Area                        ; select this window for
                                           ; PRACTICE I/O
AREA.view IO-Area                          ; open this window

PRINT "Set the PC value"                   ; print text
ENTER &pc                                  ; get value
Register.Set pc &pc                        ; set register PC

WINClear TOP                               ; delete I/O window
AREA.RESet                                 ; reset AREA system
ENDDO
```

The screen update may be controlled by the **SCREEN** commands.

| | |
|---|---|
| **SCREEN.display** | Update screen now |
| **SCREEN.ALways** | Update screen after every command line |
| **SCREEN.ON** | Update screen on print commands |
| **SCREEN.OFF** | Don't update screen as long as script is running |
| **SCREEN.WAIT** | Halt PRACTICE script execution until the data to be displayed in a window has been processed. |

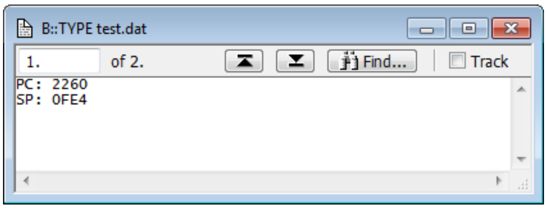
# File Operations

Test data can be written to and read from files. Files must be opened first before accessing them.

| | |
|---|---|
| **OPEN** | Open file |
| **CLOSE** | Close file |
| **READ** | Read data from file |
| **WRITE** | Write data to file |
| **APPEND** | Append data to file |

**Example 1**: A new file called test.dat is created, and register information is written to the newly created file. The result is then displayed in a **TYPE** window.

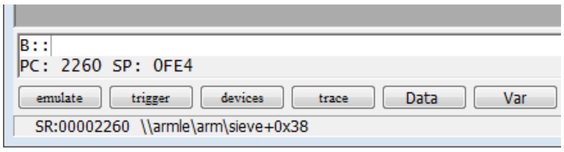
```
OPEN  #1 test.dat /Create
WRITE #1 "PC: " Register(pc)
WRITE #1 "SP: " Register(sp)
CLOSE #1
TYPE test.dat
ENDDO
```



```
PC: 2260
SP: 0FE4
```

**Example 2**: The test.dat file is opened for reading. Two lines are read from this file and stored in two PRACTICE macros, which are then printed to the TRACE32 message line.

```
OPEN  #1 test.dat  /Read
READ  #1 %LINE &pc ;line 1 of file
READ  #1 %LINE &sp ;line 2 of file
CLOSE #1
PRINT "&pc " "&sp"
ENDDO
```



```
B::
PC: 2260 SP: 0FE4
SR:00002260  \\armle\arm\sieve+0x38
```

## Automatic Start-up Scripts

When the TRACE32 software is installed, the script **autostart.cmm** is copied to the TRACE32 system directory. The **autostart.cmm** is always automatically executed after TRACE32 has booted. It provides various convenience features defined by Lauterbach.

It is recommended not to change the **autostart.cmm**, because every software update from Lauterbach will restore the file **autostart.cmm** to its default content.

The **autostart.cmm** calls the following scripts if they exist:

- **~~/system-settings.cmm**, where  ~~ represents the TRACE32 system directory.

  It is recommended to add here extra TRACE32 settings that should be available to **all users** of a TRACE32 installation. Typical extra settings are menu/toolbar extensions or user-defined dialogs.

- **UAD/user-settings.cmm**, where UAD represents the *user-specific* application data directory. The TRACE32 function **VERSION.ENVironment(UAD)** returns the path of this directory.

  Users can add all their preferred extra TRACE32 settings to the **user-settings.cmm** script. Typical extra settings are all settings of the **SETUP** command group and personal menu/toolbar extensions.

- **./work-settings.cmm**, where the leading “.” represents the working directory from where TRACE32 was started.

| | |
|---|---|
| **NOTE:** | If you don’t use the command line option **-s** *<startup_script>* and don’t have the file **autostart.cmm** either, TRACE32 will fall back to a legacy mode and execute the script **t32.cmm** from the working directory or from the TRACE32 system directory if the **t32.cmm** does not exist in the working directory. |

### What to Know about all Other Start-up Scripts

If you have a script that sets up your debug environment and if this script should be executed automatically after **autostart.cmm** is done, you can specify this script as parameter for the TRACE32 executable.

```
c:\t32\t32arm.exe -s g:\and\arm\start_up.cmm
```

It is possible to pass parameters directly to the start-up script.

```
c:\t32\t32arm.exe -s g:\and\arm\start_up.cmm param1 param2 param3
```

Parameters can be read by the start-up script as described in **“Parameter Passing”**, page 12.

The command line option **--t32-safestart** suppresses the execution of the **autostart.cmm** and of any other start-up script.

# Logging the Call Hierarchy of PRACTICE Scripts

The call hierarchy of PRACTICE scripts (*.cmm) can be logged automatically or manually. In either case, the log mechanism is based on the **LOG.DO** command.

During start-up of TRACE32, the PRACTICE script calls are always logged automatically. The log file contents are output to an autostart log file in the temporary directory of TRACE32. On every start-up of TRACE32, the previous autostart log file is overwritten and a new one is generated. The current autostart log is accessible via the **File** menu in TRACE32.

In addition, you can manually log PRACTICE script calls at any time after TRACE32 has started. When you initiate the log, you can choose folder and file name. To display the log file, use the **TYPE** or the **EDIT** command on the TRACE32 command line.

To enable logging, use one of the following options:

- **autostart.cmm**: On start-up of TRACE32, the autostart.cmm is called automatically, and the **LOG.DO** command in the autostart.cmm generates the autostart log file.
- **--t32-logautostart**: This command line option starts **LOG.DO** internally to generate an autostart log file.
  - This option is only needed (a) if you do not have the autostart.cmm or (b) if the script block with the **LOG.DO** command has been deleted from the autostart.cmm.
  - For a description of the command line options and an example for **--t32-logautostart**, refer to **“Command Line Arguments for Starting TRACE32”** in TRACE32 Installation Guide, page 53 (installation.pdf).
    Tip: To explicitly disable all PRACTICE script calls on start-up, use **--t32-safestart**.
- **TRACE32 command line**: Use the **LOG.DO** *<file>* command to generate your log file.

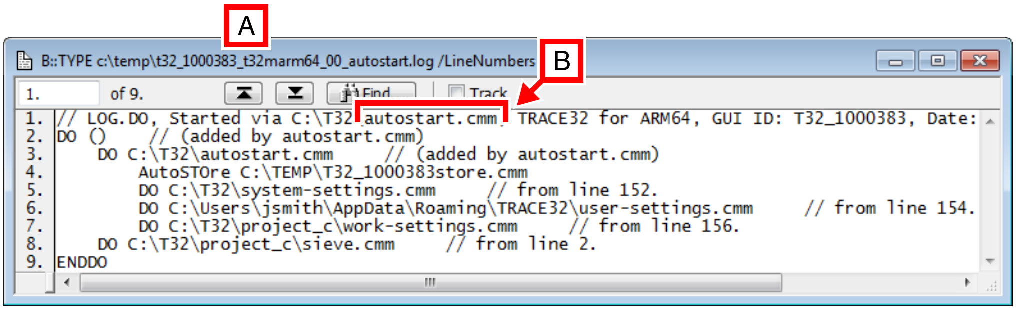
**To access the autostart log file in TRACE32:**

1. Start TRACE32 via T32Start.

   The autostart.cmm generates the autostart log file automatically.

2. Choose **File** menu > **Automatic Scripts on Start** > **View Autostart log**.

   The file opens in the **TYPE** window. The screenshot shows an example of an autostart log file:



```
B::TYPE c:\temp\t32_1000383_t32marm64_00_autostart.log /LineNumbers
```

```
// LOG.DO, Started via C:\T32\autostart.cmm, TRACE32 for ARM64, GUI ID: T32_1000383, Date:
DO ()    // (added by autostart.cmm)
    DO C:\T32\autostart.cmm     // (added by autostart.cmm)
        AutoSTOre C:\TEMP\T32_1000383store.cmm
        DO C:\T32\system-settings.cmm     // from line 152.
        DO C:\Users\jsmith\AppData\Roaming\TRACE32\user-settings.cmm     // from line 154.
        DO C:\T32\project_c\work-settings.cmm     // from line 156.
    DO C:\T32\project_c\sieve.cmm     // from line 2.
ENDDO
```

**A** The file name convention of the autostart log is described below.

**B** The log file header tells you how the autostart log was generated.

**File name convention of the autostart log file**: ~~~/*<id>*_t32m*<arch>*_*<xx>*_autostart.log

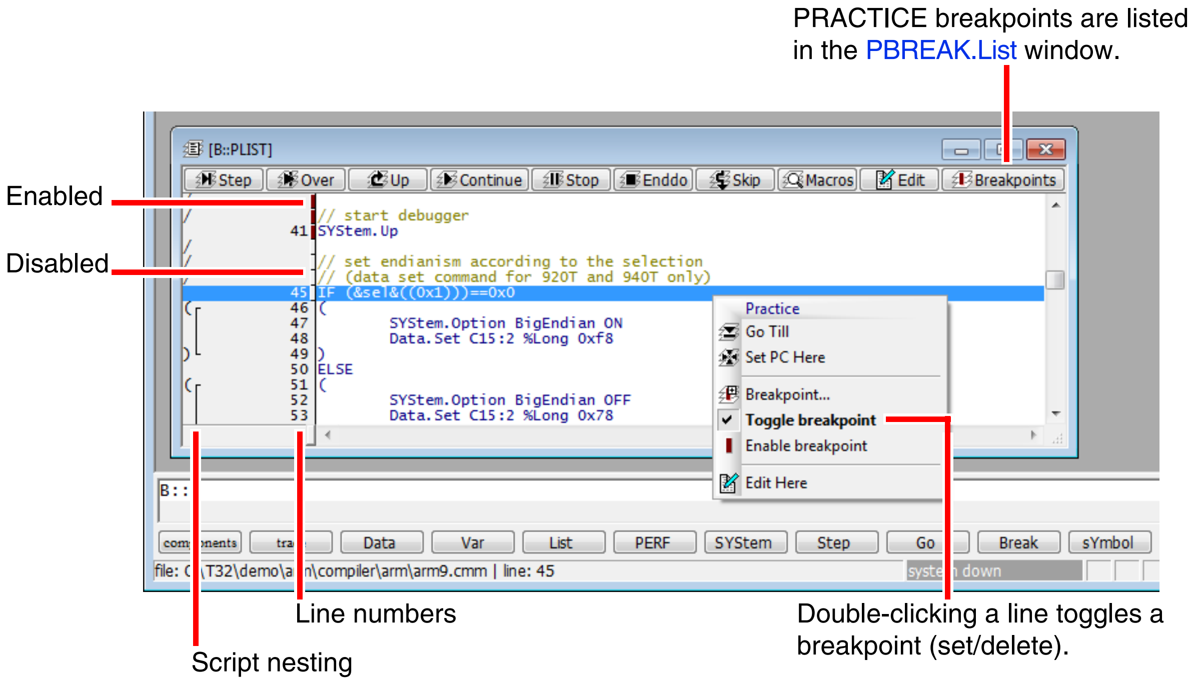
| | |
|---|---|
| ~~~ | Path prefix of the temporary directory of TRACE32. See also **OS.PresentTemporaryDirectory()**. |
| *<id>* | ID of the PowerView GUI that was started. See also **OS.ID()**. |
| t32m*<arch>* | Name of the PowerView executable (without file extension), e.g. “t32marm” |
| *<xx>* | The instance number of the PowerView executable. |

# Debugging PRACTICE Scripts

TRACE32 supports extensive debugging features for PRACTICE scripts. The **PEDIT** command allows you to create and edit PRACTICE scripts. Two basic windows show the script, the internal stack, and the PRACTICE macros.

| | |
|---|---|
| **PEDIT** | Edit PRACTICE scripts |
| **PLIST** | List PRACTICE script |
| **PMACRO.list** | List PRACTICE script nesting and PRACTICE macros |

In a **PLIST** window, you can set an unlimited number of program breakpoints to debug your PRACTICE scripts. Double-clicking an entire line sets a breakpoint; double-clicking the same line again removes the breakpoint. That is, you can toggle breakpoints by double-clicking a line.
Alternatively, right-click a line, and then select **Toggle breakpoint** from the popup menu.



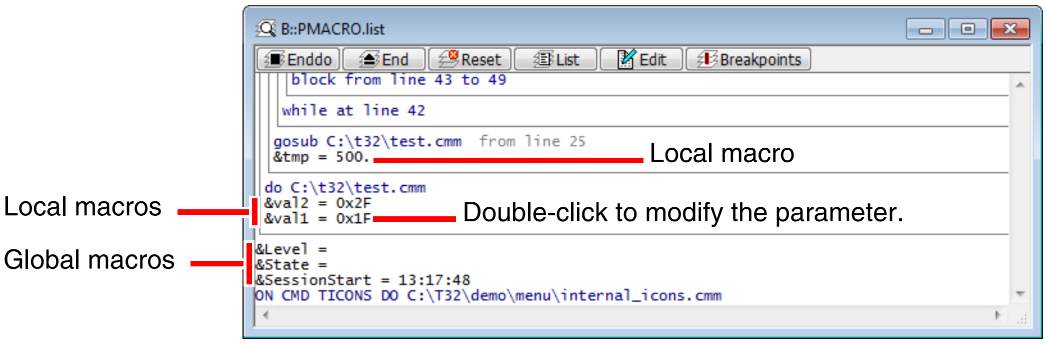
```
   // start debugger
41 SYStem.Up

   // set endianism according to the selection
   // (data set command for 920T and 940T only)
45 IF (&sel&((0x1)))==0x0
46 (
47         SYStem.Option BigEndian ON
48         Data.Set C15:2 %Long 0xf8
49 )
50 ELSE
51 (
52         SYStem.Option BigEndian OFF
53         Data.Set C15:2 %Long 0x78
```

file: C:\T32\demo\arm\compiler\arm\arm9.cmm | line: 45

Enabled breakpoints are flagged with a small red bar, disabled breakpoints are flagged with a small gray bar in the **PLIST** window.

| | |
|---|---|
| **PBREAK.Set** | Set breakpoints in PRACTICE scripts |
| **PBREAK.Delete** | Delete breakpoints |
| **PBREAK.List** | Display breakpoint list |
| **PBREAK.ENable** | Enable breakpoint |
| **PBREAK.DISable** | Disable breakpoint |

The **PMACRO.list** window shows the script nesting, the local and global PRACTICE macros and the ON and GLOBALON definitions:



```
block from line 43 to 49
while at line 42
gosub C:\t32\test.cmm  from line 25
&tmp = 500.
do C:\t32\test.cmm
&val2 = 0x2F
&val1 = 0x1F
&Level =
&State =
&SessionStart = 13:17:48
ON CMD TICONS DO C:\T32\demo\menu\internal_icons.cmm
```

Double-clicking a PRACTICE macro (e.g. `&val1`) inserts it into the TRACE32 command line, where you can modify the parameter of the PRACTICE macro.

Scripts can be executed step by step with the **PSTEP** command. The **Stop** button in the TRACE32 main toolbar stops any running PRACTICE script.

| | |
|---|---|
| **PSTEP** *<script>* | Start script to be debugged in single step mode |

```
PBREAK.Set 4. test.cmm          ; set breakpoint
DO test.cmm                     ; run till line 4
PSTEP
PSTEP                           ; single step
```

# Appendix A

**In this appendix:**

- Here are a few suggestions how to run demo scripts you have copied from the pdf manuals:
  - Create PRACTICE script on the fly [more].
  - Create a permanent toolbar button for a `test.cmm` [more].
  - Run a copied script as an embedded script [more].
- Run a demo script from the TRACE32 ~~\demo folder [more].

## How to Run Demo Scripts Copied from the PDF Manuals

The pdf manuals provide a few read-to-run demo scripts (in addition to the PRACTICE script segments). One way to try these demo scripts is to create a `test.cmm` in the **PEDIT** window, copy and paste the demo script into the *.cmm file, and then execute it.

For how-to information, see step-by-step procedure below.

### Create a PRACTICE Script on the Fly

1. At the TRACE32 command line, type:

```
PEDIT ~~~/test.cmm
```

   The PRACTICE script editor **PEDIT** opens, displaying the `test.cmm`. The path prefix `~~~` expands to the temporary directory of TRACE32.

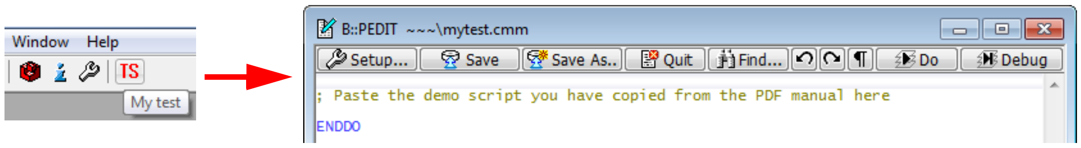
2. Paste the demo script that you have copied from a manual into the `test.cmm`.
3. Append **ENDDO** at the end of the demo script (if **ENDDO** is missing from the demo script).
4. Click **Save**.
5. To step through line by line, click **Debug** in the **PEDIT** window,
   and then **Step** in the **PLIST** window.
6. To execute the PRACTICE script file, click **Do** in the **PEDIT** window.
7. Optionally, click **Macro** to view the PRACTICE stack frame.

## Create a Permanent Toolbar Button for mytest.cmm

1. Add the following to `c:\t32\system-settings.cmm` (create the file if it does not yet exist):

```
MENU.AddTool "My test" "TS,R" "PEDIT ~~~/mytest.cmm"
```

2. Restart TRACE32.
3. Click the new toolbar button to open the PRACTICE script editor.



```
; Paste the demo script you have copied from the PDF manual here
ENDDO
```

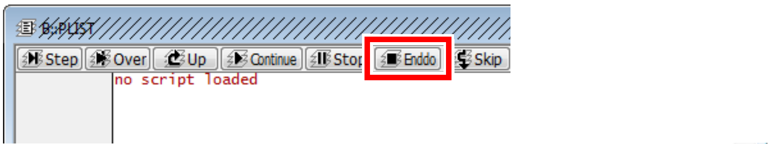
## Run a Demo Script as an Embedded Script

To reproduce the following step-by-step procedure, we suggest that you copy this demo script:

```
;set a test pattern to the virtual memory of TRACE32
Data.Set VM:0--0x4f %Byte 1 0 0 0
Data.dump VM:0x0 ;open the Data.dump window
;visualize the contents of the TRACE32 virtual memory as a graph
Data.DRAWFFT %Decimal.Byte VM:0++0x4f 2.0 512.
```

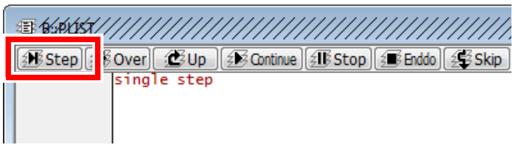
**To run a demo script as an embedded script:**

1. Copy the demo script from the pdf manual.
2. At the TRACE32 command line, type **`PLIST`** to open a **PLIST** window.
3. Click **Enddo** until the **PLIST** window displays *no script loaded*.



```
no script loaded
```

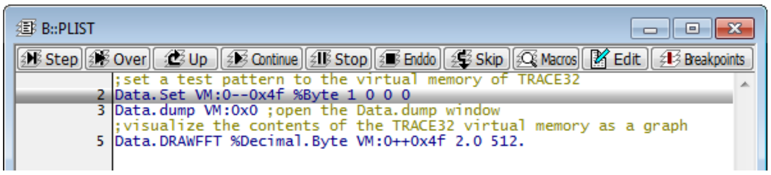
4. Click **Step**. The **PLIST** window displays *single step*.



```
single step
```

5. Paste the demo script into the TRACE32 command line.

Result:



```
  ;set a test pattern to the virtual memory of TRACE32
2 Data.Set VM:0--0x4f %Byte 1 0 0 0
3 Data.dump VM:0x0 ;open the Data.dump window
  ;visualize the contents of the TRACE32 virtual memory as a graph
5 Data.DRAWFFT %Decimal.Byte VM:0++0x4f 2.0 512.
```

6. Do one of the following:
   - Click **Continue** to run the embedded script.
   - Click **Step** to step through line by line.

## Demo Scripts in the TRACE32 Demo Folder

Using the **Search for scripts** window in TRACE32, you can search for PRACTICE demo scripts (*.cmm) in your local TRACE32 demo folder as well as for the newest scripts on the Lauterbach website.

The **Search for scripts** window displays a brief description for each script you have selected. The descriptions are extracted from the metadata in the script header. Double-clicking a script lets you preview the source code of a script before running it.

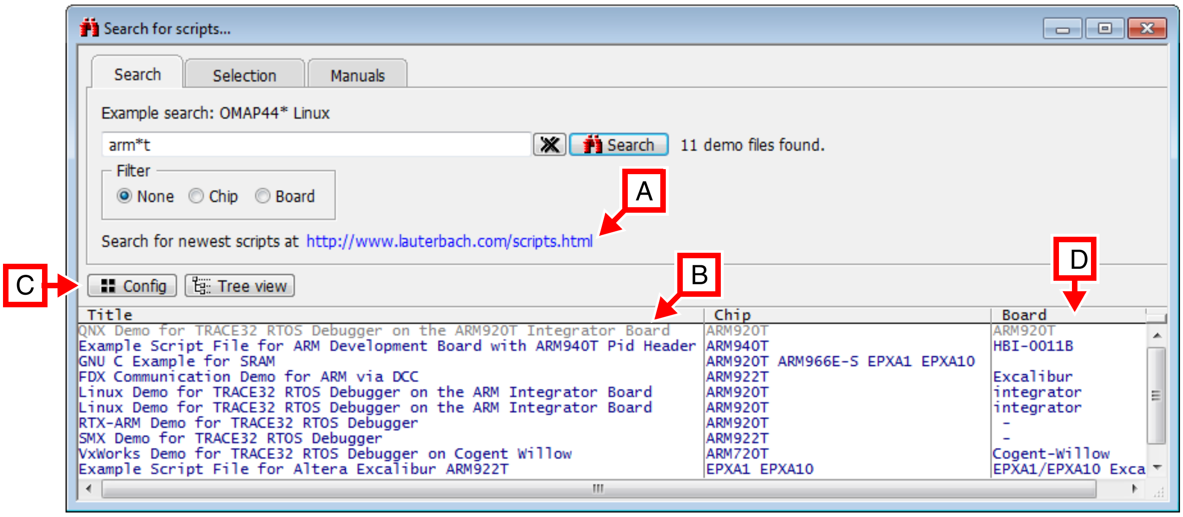
**To search for demo scripts in your local TRACE32 demo folder:**

1. Choose **File** menu > **Search for Script** to open the window of the same name,
   or at the TRACE32 command line type: **WELCOME.SCRIPTS**

2. Under **Example search**, enter what you are looking for, e.g. a chip or board name.
   The wildcard (*) is supported.

3. Optionally, set the filter to **Chip** or **Board**.

4. Click **Search**.
   The demo scripts meeting the search criteria are listed at the bottom of the window.



Example search: OMAP44* Linux

arm*t

11 demo files found.

Filter: [✓ None] Chip Board

Search for newest scripts at http://www.lauterbach.com/scripts.html

```
Title                                                       Chip                         Board
QNX Demo for TRACE32 RTOS Debugger on the ARM920T Integrator Board   ARM920T             ARM920T
Example Script File for ARM Development Board with ARM940T Pid Header ARM940T            HBI-0011B
GNU C Example for SRAM                                      ARM920T ARM966E-S EPXA1 EPXA10
FDX Communication Demo for ARM via DCC                      ARM922T                      Excalibur
Linux Demo for TRACE32 RTOS Debugger on the ARM Integrator Board     ARM920T             integrator
Linux Demo for TRACE32 RTOS Debugger on the ARM Integrator Board     ARM920T             integrator
RTX-ARM Demo for TRACE32 RTOS Debugger                      ARM920T                       -
SMX Demo for TRACE32 RTOS Debugger                          ARM922T                       -
VxWorks Demo for TRACE32 RTOS Debugger on Cogent Willow     ARM720T                      Cogent-Willow
Example Script File for Altera Excalibur ARM922T            EPXA1 EPXA10                 EPXA1/EPXA10 Exca
```

**A** Click the hyperlink to continue your search at **www.lauterbach.com/scripts.html**.

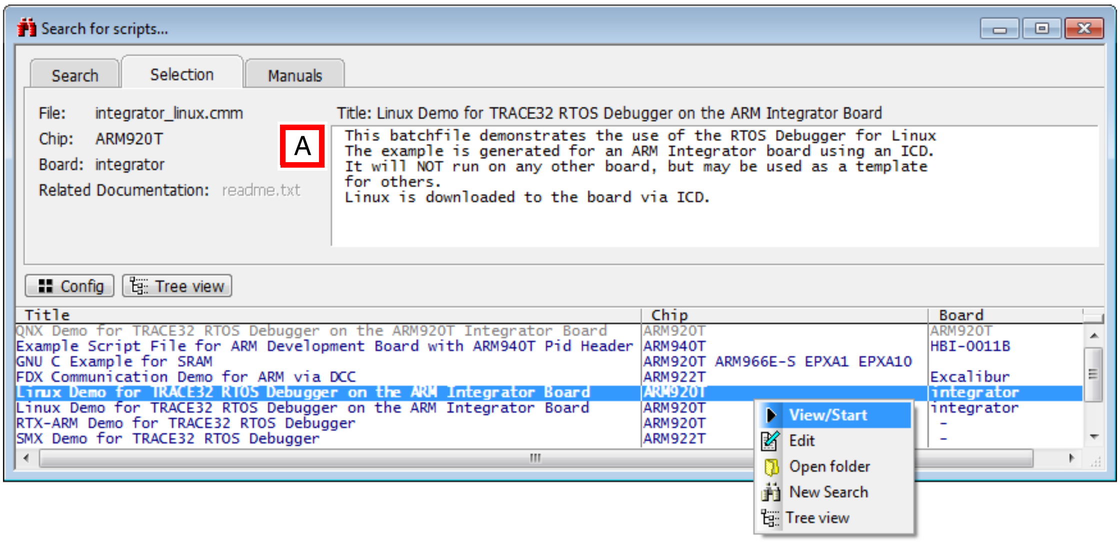
**B** PRACTICE script templates are highlighted in gray, ready-to-run scripts are highlighted in blue.

**C** Allows you to add your own search paths to the demo script search.

**D** Clicking the column header changes the sort order.

**To preview and run a demo script:**

1. Click a script to automatically switch to the **Selection** tab, where you can view a brief description of the selected script [**A**].



File: integrator_linux.cmm
Chip: ARM920T
Board: integrator
Related Documentation: readme.txt

Title: Linux Demo for TRACE32 RTOS Debugger on the ARM Integrator Board

```
This batchfile demonstrates the use of the RTOS Debugger for Linux
The example is generated for an ARM Integrator board using an ICD.
It will NOT run on any other board, but may be used as a template
for others.
Linux is downloaded to the board via ICD.
```

| Title | Chip | Board |
|---|---|---|
| QNX Demo for TRACE32 RTOS Debugger on the ARM920T Integrator Board | ARM920T | ARM920T |
| Example Script File for ARM Development Board with ARM940T Pid Header | ARM940T | HBI-0011B |
| GNU C Example for SRAM | ARM920T ARM966E-S EPXA1 EPXA10 | |
| FDX Communication Demo for ARM via DCC | ARM922T | Excalibur |
| Linux Demo for TRACE32 RTOS Debugger on the ARM Integrator Board | ARM920T | integrator |
| Linux Demo for TRACE32 RTOS Debugger on the ARM Integrator Board | ARM920T | integrator |
| RTX-ARM Demo for TRACE32 RTOS Debugger | ARM920T | - |
| SMX Demo for TRACE32 RTOS Debugger | ARM922T | - |

2. To view the source code of a script in the **PSTEP** window, double-click the script or right-click, and then click **View/Start** from the popup menu.

3. To run the script, click **Continue** in the **PSTEP** window.

| Popup Menu - Option | Description |
|---|---|
| **View/Start** | Opens the script in the **PSTEP** window. |
| **Edit** | Opens the script in the **PEDIT** window. |
| **Open Folder** | Opens the file explorer and selects the script file. |
| **New Search** | Switches to the Search tab, where you can start a new search. |
| **Tree view** | Displays an file-explorer-like tree view of the folders and demo scripts. |