スクリプト言語 PRACTICE トレーニング

TRACE32 Online Help
TRACE32 Directory
TRACE32 Index
TRACE32 Training .....
スクリプト言語 PRACTICE トレーニング 1
チュートリアル動画 4
スクリプトの入手 4
スクリプト言語 PRACTICE について 5
スクリプトの用途 5
スクリプトの実行方法 6
PRACTICE スクリプトの作成方法 8
TRACE32 の設定項目をスクリプトに変換 8
LOG コマンド 13
Command History 14
スクリプトエディタ PEDIT 15
シンタックスハイライト 17
PRACTICE スクリプトのデバッグ 19
デバッグ環境 20
PRACTICE スタックの表示 24
PRACTICE 言語 25
プログラム要素 25
コメント 25
コマンド 25
関数 26
ラベル 27
PRACTICE フロー制御 28
条件実行 28
コマンドリスト 28
サブルーチンコール 33
コマンドリスト 33
例 34
GOTO/JUMPTO 文 36
コマンドリスト 36
例 37
スクリプトの呼び出し 39
コマンドリスト 39
例 39
PRACTICE マクロ 40
マクロの宣言 40
マクロへのコンテンツの割り当て 42
マクロのハンドリング 44
文字列としてのマクロ 46

©1989-2020 Lauterbach GmbH

Macros as Numbers 48
マクロの動作確認 51
複雑なデータ構造 52
スクリプト例 53
プログラムの実行とテストレポートの生成 53
アドレスの内容を確認 59
アドレス範囲の内容を確認 60
変数の内容の確認 63
書式を指定して変数を記録 64
変数を CSV ファイルに記録 65
関数の機能テスト 67
パラメータファイルを利用した関数の機能テスト 68
パラメータの受け渡し 70
PRACTICE スクリプト / 関数へのパラメータの受け渡し 70
ENTRY コマンドと PARAMETERS/RETURNVALUES コマンド 78
ホストのオペレーティングシステム 83
オペレーティングシステムの検出 83
結果の出力 84
環境変数の参照 86
OS コマンドの実行 87
ファイル操作 88
時間と日付 92
I/O コマンド 93
出力コマンド 93
入力コマンド 93
AREA ウィンドウ経由の入出力 95
PRACTICE によるイベント制御 97
ダイアログ 98
ダイアログのプログラム 101
サイズ、位置の制御 104
Control Properties 106
コントロールの有効 / 無効化 107
コントロールからのデータ取得 108
コントロールへの値 / 状態の設定 111
コマンドの実行 112
ファイルのブラウジング 114
Icons 116
ダイアログの例 117
マルチコア環境での PRACTICE の利用 122
インスタンス間の通信 123
ロバスト性を意識した PRACTICE スクリプトの設計 125
ディレクトリパス名の扱い 126
ホスト OS 129
デバッガハードウェア 130
ターゲット CPU/ ボード 131
TRACE32 PowerView バージョン 133

©1989-2020 Lauterbach GmbH

TRACE32 設定 133
設定の保存 / 取得 135
エラーハンドリング 138
引数の扱い 140
独自コマンドの作成 141
間違いやすい記述 142

©1989-2020 Lauterbach GmbH

スクリプト言語 PRACTICE トレーニング

Version 15-Jun-2020

チュートリアル動画

スクリプト言語 PRACTICE のチュートリアル動画は、以下の URI で参照可能です。

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

スクリプトの入手

ローターバッハ社は、以下の URI で実践的な PRACTICE スクリプトを公開し、日々更新しています。

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

©1989-2020 Lauterbach GmbH

スクリプト言語 PRACTICE について

スクリプトの用途

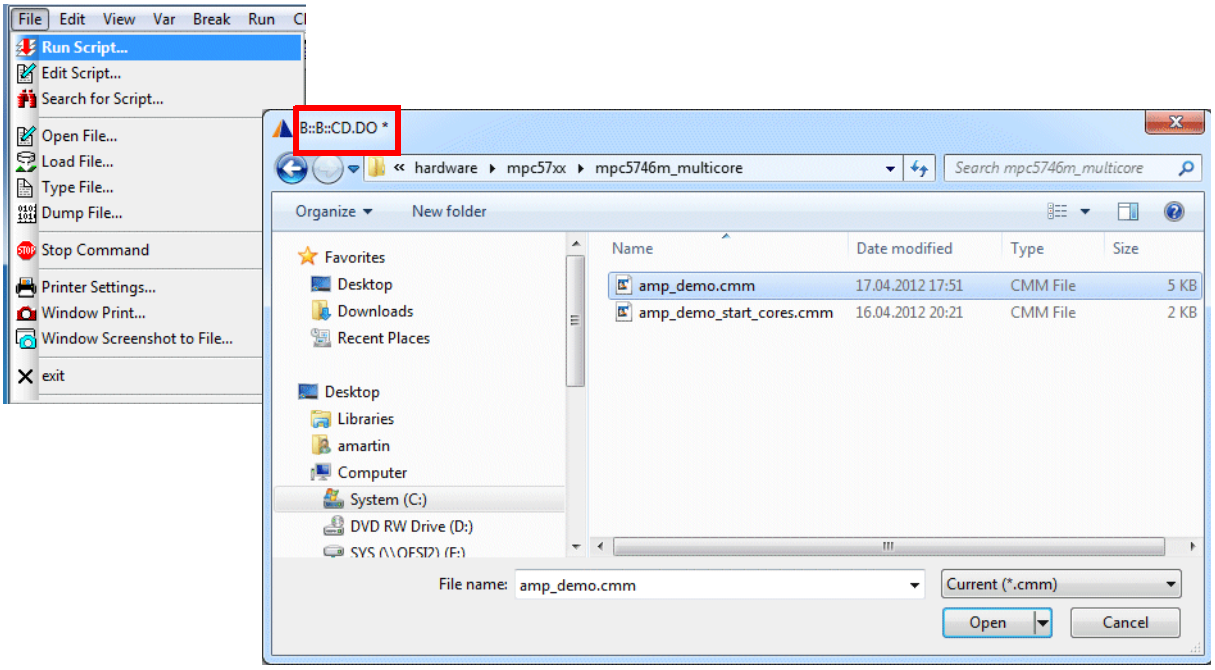
スクリプト言語 PRACTICE の主な用途と機能を以下に示します。

- デバッガツールの起動シーケンスの提供
- フラッシュメモリへの書き込みの自動化
- デバッガユーザインターフェースのカスタマイズ
- TRACE32 の設定の保存、再読み込み
- 自動テストの実行

PRACTICE の拡張子の標準は **.cmm** になります。

©1989-2020 Lauterbach GmbH

スクリプトの実行方法




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

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

ChDir.DO *<ファイル名>*	スクリプト *<ファイル名>* のあるディレクトリに移動し、スクリプトを起動。
DO *<ファイル名>*	*<ファイル名>* で指定されたスクリプトを実行。
PATH [+] *<パス名>*	PRACTICE スクリプトをサーチするパスの定義。

```
DO memtest

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

PATH c:/t32/tests
```

©1989-2020 Lauterbach GmbH

TRACE32 の起動時に任意の PRACTICE スクリプトが実行可能です。詳細は PRACTICE Script Language User's Guide (practice_user.pdf) "Automatic Start-up Scripts" を参照してください。

©1989-2020 Lauterbach GmbH

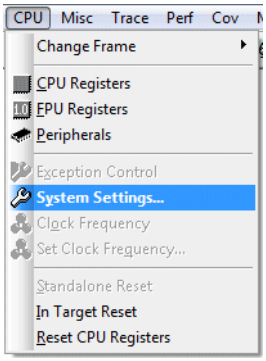
TRACE32 の設定項目をスクリプトに変換

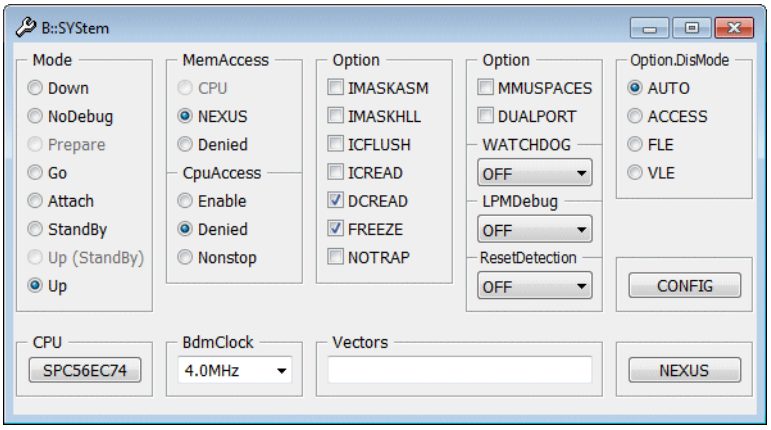
STOre コマンドと **ClipSTOre** コマンドは、指定された TRACE32 〈***設定***〉をいつでも再設定可能とするスクリプトを生成します

〈*設定*〉は通常コマンドグループによる設定になります。

〈*設定*〉	
SYStem	コマンドグループ SYStem の設定
Break	コマンドグループ Break の設定
Win	TRACE32 ウィンドウ設定

©1989-2020 Lauterbach GmbH

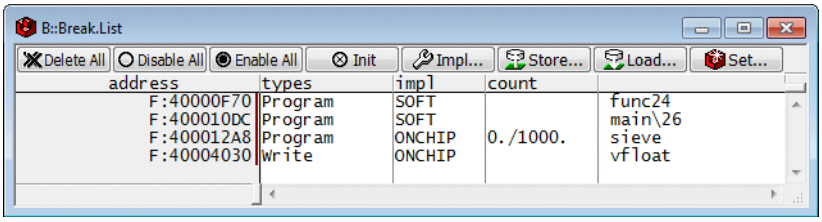





```
ClipSTOre SYStem
```

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

©1989-2020 Lauterbach GmbH

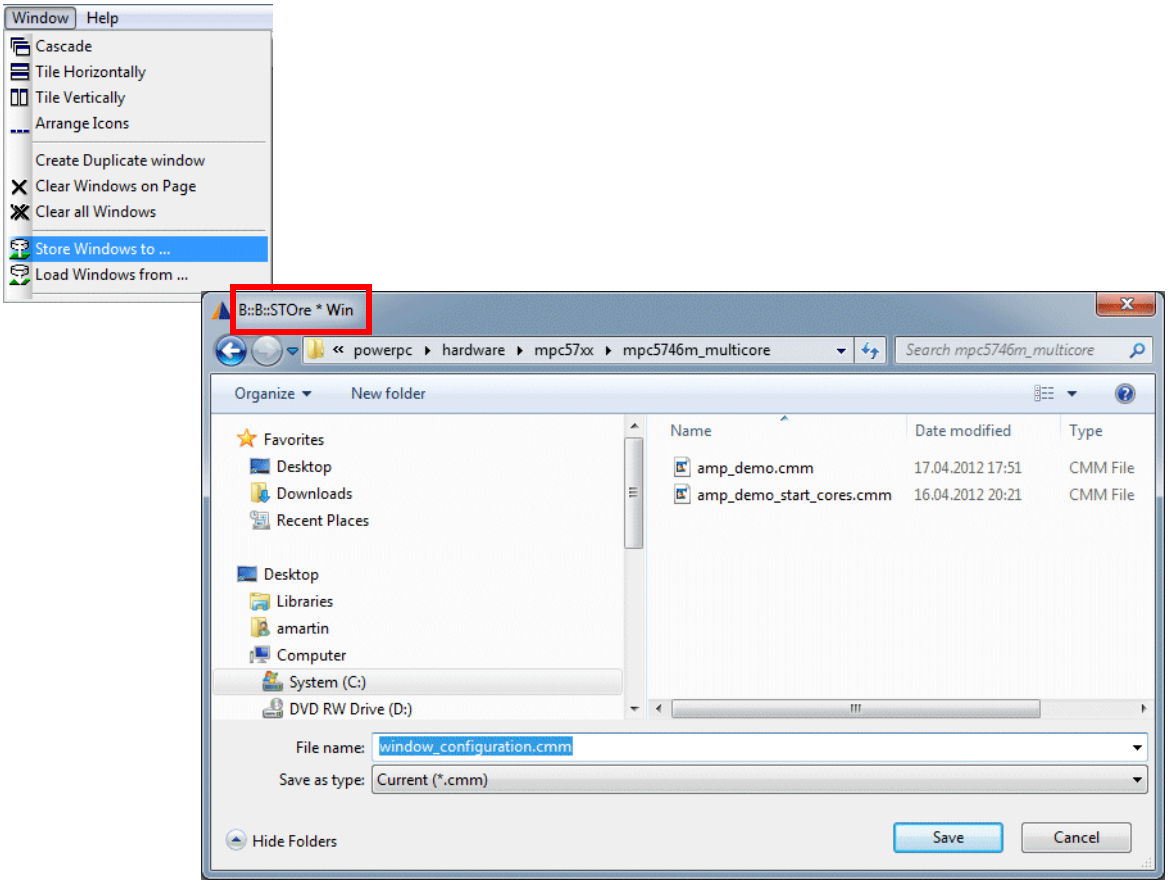



```
ClipSTOre Break
```

```
Break.RESet
Break.Set      func24 /Program
Break.Set      main\38 /Program
Break.Set      sieve /Program /Onchip /COUNT 1000.
Var.Break.Set mstatic1; /Write
```

デフォルトでは、ブレークポイントはシンボル名で保存されます。

©1989-2020 Lauterbach GmbH




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

©1989-2020 Lauterbach GmbH

```
B::

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

WinPAGE.Create P000
WinCLEAR

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

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

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

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

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

WinPAGE.select P000

ENDDO
```

.



すべての PRACTICE スクリプトは、**ENDDO** 命令で終了する必要があります。

ENDDO	PRACTICE スクリプトからリターン。

©1989-2020 Lauterbach GmbH

LOG コマンド

LOG コマンドを利用して、PowerView GUI 内でのほとんどの実行記録を生成することが可能です。

LOG コマンドの制御

LOG.OPEN *<log_file>*	コマンド LOG 用のファイルの生成、オープン。(ファイルのデフォルトの拡張子は **.log** です。)
LOG.CLOSE	コマンド LOG ファイルのクローズ。
LOG.OFF	コマンド LOG を一時的に記録停止。
LOG.ON	コマンド LOG のスイッチを記録開始。
LOG.type	記録中のコマンド LOG を表示。

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

LOG.type               // Displays .log file contents while recording

…                      // Recording

LOG.CLOSE              // Closes .log file
```

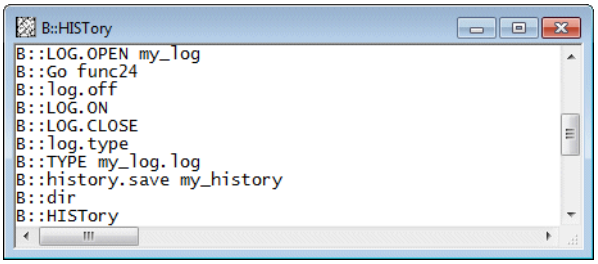
コマンドログの内容例

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

©1989-2020 Lauterbach GmbH

	コマンド履歴には、コマンドラインに入力されたコマンドのみが記録されます。コマンド履歴ファイルのデフォルトの拡張子は (.log) です。

HISTory.type コマンド履歴の表示



HISTory.SAVE [*<ファイル>*] コマンド履歴の保存

HISTory.SIZE [*<サイズ>*] コマンド履歴のサイズの定義

デフォルトでは、スクリプト autostart.cmm には改行が含まれています。

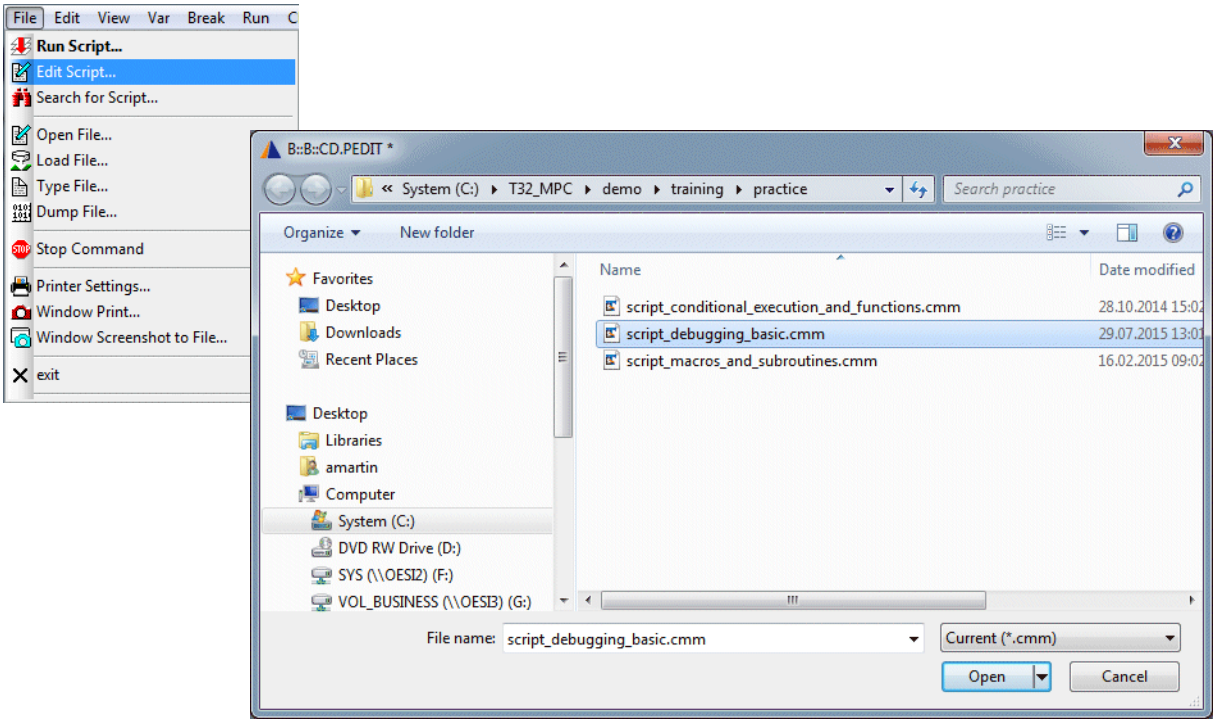
```
AutoSTOre , HISTory
```

上記では、TRACE32 終了時にコマンド履歴をテンポラリディレクトリに自動的に保存し、TRACE32 の再起動時にコマンド履歴を呼び出します。

AutoSTOre *ファイル名* >[*<設定>*] TRACE32 終了時に定義した設定を自動的に保存し、TRACE32 再起動時に再有効化

©1989-2020 Lauterbach GmbH

スクリプトエディタ PEDIT

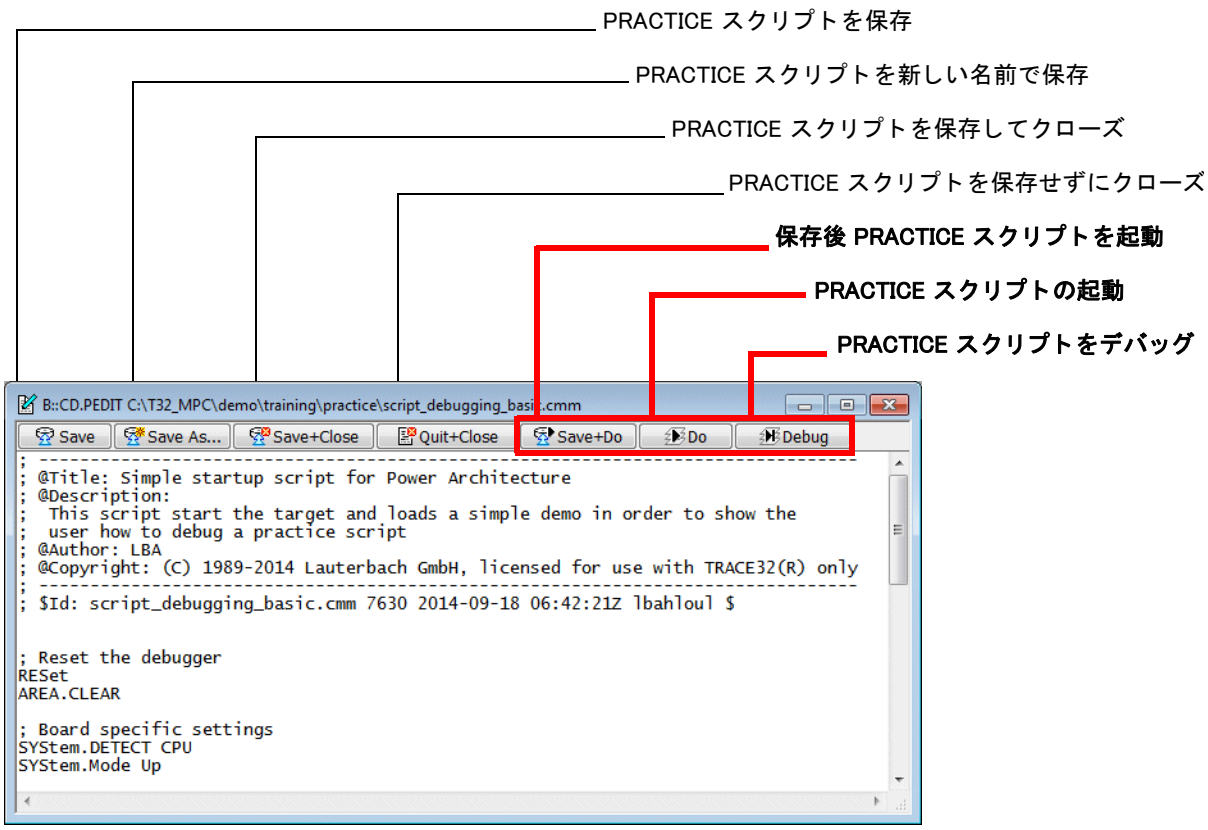



```
CD.PEDIT *                      // "*" opens a file browser for script
                                // selection

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

ChDir.PEDIT *〈ファイル名〉*	スクリプト *〈ファイル名〉* の配置されているディレクトリに移動し、スクリプトエディタ PEDIT でスクリプトをオープン。
PEDIT *〈ファイル名〉*	スクリプトエディタ PEDIT でスクリプト *〈ファイル名〉* をオープン。

©1989-2020 Lauterbach GmbH

PEDIT は、標準的なエディタ機能と、スクリプトの起動、デバッグが行えます。



©1989-2020 Lauterbach GmbH

シンタックスハイライト

PRACTICE スクリプトエディタ **PEDIT** は、シンタックスハイライト、自動インデント、アンドゥ、リドゥの機能を備えています。

内蔵の PRACTICE スクリプトエディタ PEDIT に加えて、外部エディタも利用可能です。外部エディタで PRACTICE スクリプトのシンタックスハイライトを設定するには、以下の手順で行ってください。

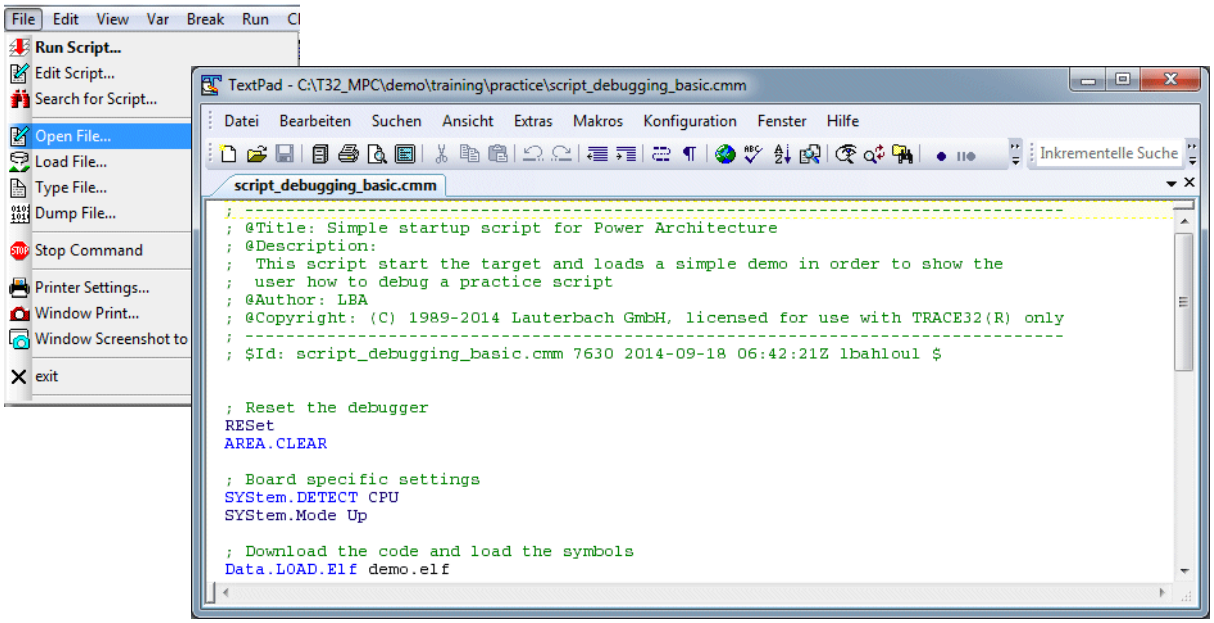
1. コマンド **SETUP.EDITEXT** を利用して、TRACE32 のエディタ EDIT の呼び出しを外部エディタに設定します。
2. ローターバッハ社が提供している各種エディタ用シンタックスハイライトファイルをインストールします。

EDIT *<ファイル名>*	標準の TRACE32 エディタでファイルをオープン。
SETUP.EDITEXT ON *<コマンド>*	EDIT コマンドにより、指定された外部エディタを呼び出すように設定。 *<コマンド>*は、外部エディタを起動するために TRACE32 がホスト OS に送信するコマンドです。この文字列では、以下のように置換されます。 • * はファイル名に置換 • # は行番号に置換。

ローターバッハ社は、一般的なテキストエディタ用のシンタックスハイライト設定ファイルを提供しています。
~~/demo/practice/syntaxhighlighting を参照してください。また、~~ は、Trace32 がインストールされているディレクトリを示します。デフォルトのインストールディレクトリは、c:/T32 です。

©1989-2020 Lauterbach GmbH

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



©1989-2020 Lauterbach GmbH

PRACTICE スクリプトのデバッグ

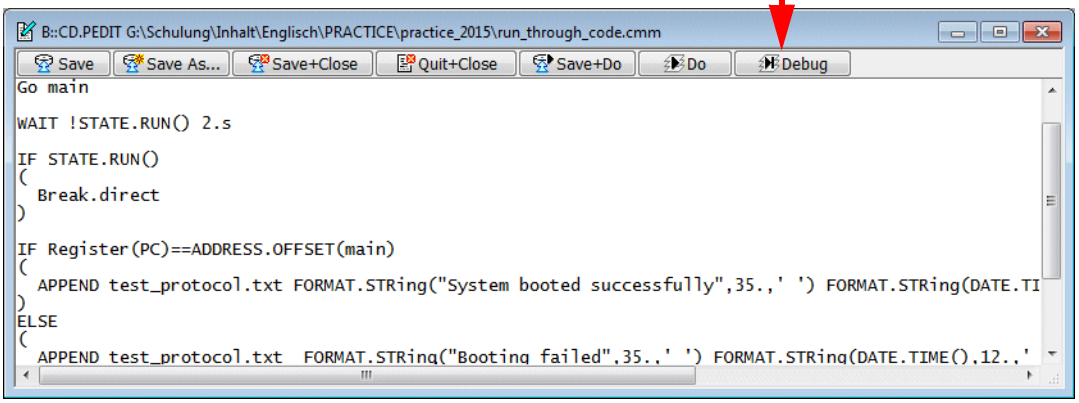
TRACE32 は、PRACTICE スクリプトのデバッグ機能を持ちます。

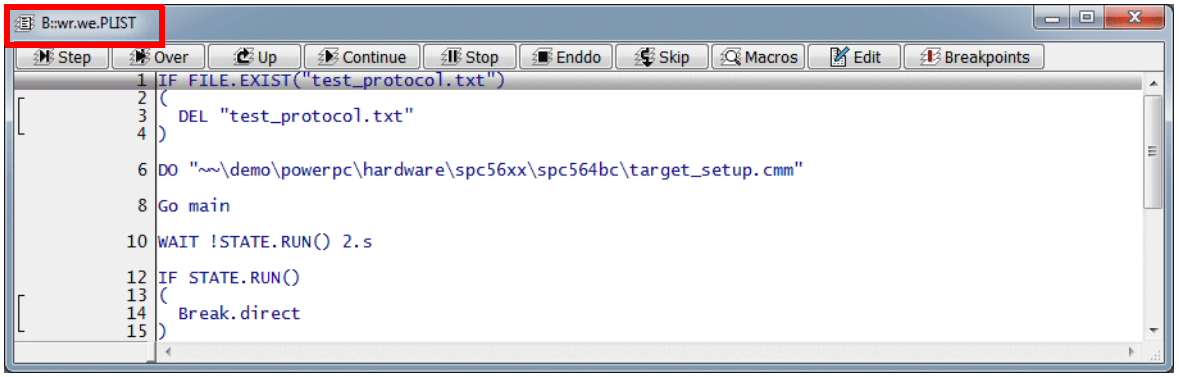
以下の URI に PRACTICE のデバッグチュートリアルビデオを公開しています。

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

©1989-2020 Lauterbach GmbH

PRACTICE スクリプトのデバッガを起動するには Debug を押下します





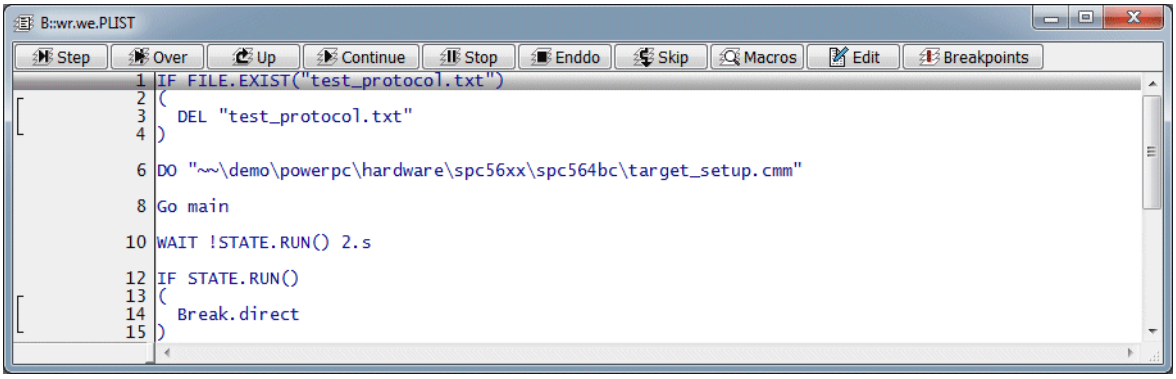
ウィンドタイトルの説明

WinResist.WinExt.PLIST	現在ロードされている PRACTICE スクリプトの表示。 WinResist: PEDIT ウィンドウは、WinCLEAR コマンドでは閉じることができません。 WinExt: TRACE32 が MDI で動作している場合でも、メインウィンドウから PEDIT ウィンドウを独立表示します。

©1989-2020 Lauterbach GmbH

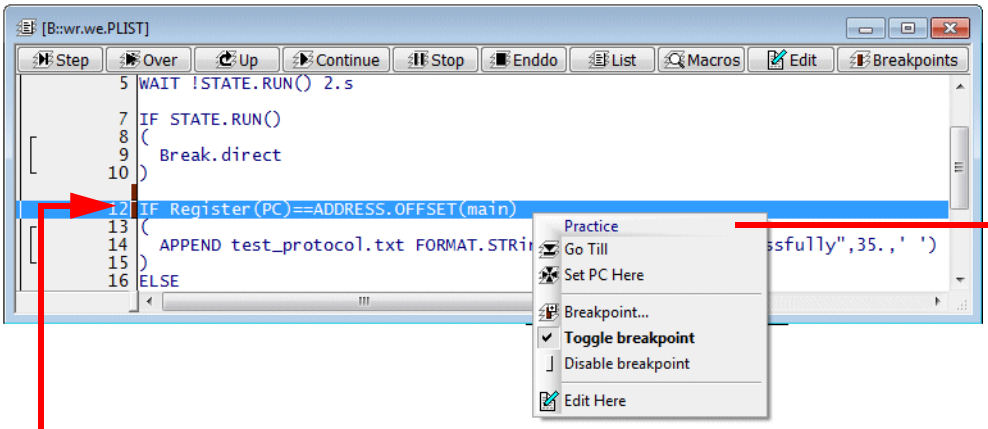
PSTEP 〈*ファイル名*〉	PRACTICE デバッガでスクリプトを起動。
ChDir.PSTEP 〈*ファイル名*〉	スクリプトが配置されているディレクトリに移動後、PRACTICE デバッガでスクリプトを起動。
WinResist.WinExt.ChDir.PSTEP 〈*ファイル名*〉	

©1989-2020 Lauterbach GmbH



PLIST/PSTEP ウィンドウのローカルボタン	
Step	PRACTICE スクリプトをシングルステップ実行します。(**PSTEP コマンド**)
Over	PRACTICE スクリプト、または PRACTICE サブルーチンを実行します。(**PSTEPOVER コマンド**)
Up	現在の PRACTICE スクリプト、またはサブルーチンを終了し、呼び出し元に戻ります。(**PSTEPOUT コマンド**)
Continue	PRACTICE スクリプトの実行を継続します。(**CONTinue コマンド**)
Stop	PRACTICE スクリプトの実行を停止します。(**STOP コマンド**)
Enddo	現在実行中の PRACTICE スクリプトを終了します。呼び出し元の PRACTICE スクリプトまで実行は継続されます。呼び出し元のスクリプトが存在しない場合は、PRACTICE スクリプトの実行を終了します。(**ENDDO コマンド**)
Skip	現在のコマンドまたはブロックをスキップします。(**PSKIP コマンド**)
Macros	PRACTICE スタックを表示します。(**PMACRO.list コマンド**)
Edit	PRACTICE エディタ PEDIT を開き、PRACTICE スクリプトを編集します。(**WinResist.WinExt.PEDIT** コマンド)
Breakpoints	**PBREAK.List** ウィンドウを開き、すべての PRACTICE ブレークポイントを表示します。

©1989-2020 Lauterbach GmbH



PRACTICE ブレークポイントはこの行に設定されています。

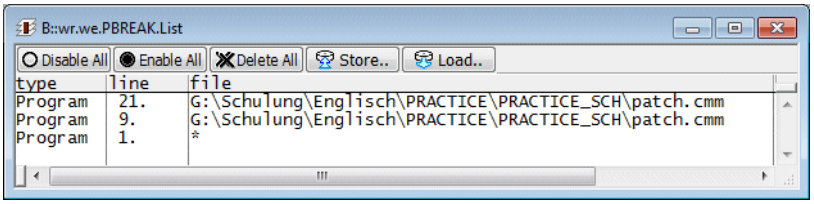
右クリックで、PRACTICE デバッグプルダウンを表示します。

PRACTICE デバッグプルダウン	
Goto Till	選択された行まで PRACTICE スクリプトを実行します。(CONTinue *〈/行番号〉コマンド*).
Set PC Here	選択された行に PRACTICE の PC を設定します。
Breakpoint ...	PBREAK.Set ダイアログを開き、PRACTICE ブレークポイントを設定します。
Toggle breakpoint	PRACTICE ブレークポイントをトグルします。
Disable breakpoint	PRACTICE ブレークポイントを無効にします。(PBREAK.DISable コマンド).
Edit Here	PRACTICE エディタを開き PRACTICE スクリプトを編集します。カーソルは自動的に選択された行に設定されます。

©1989-2020 Lauterbach GmbH

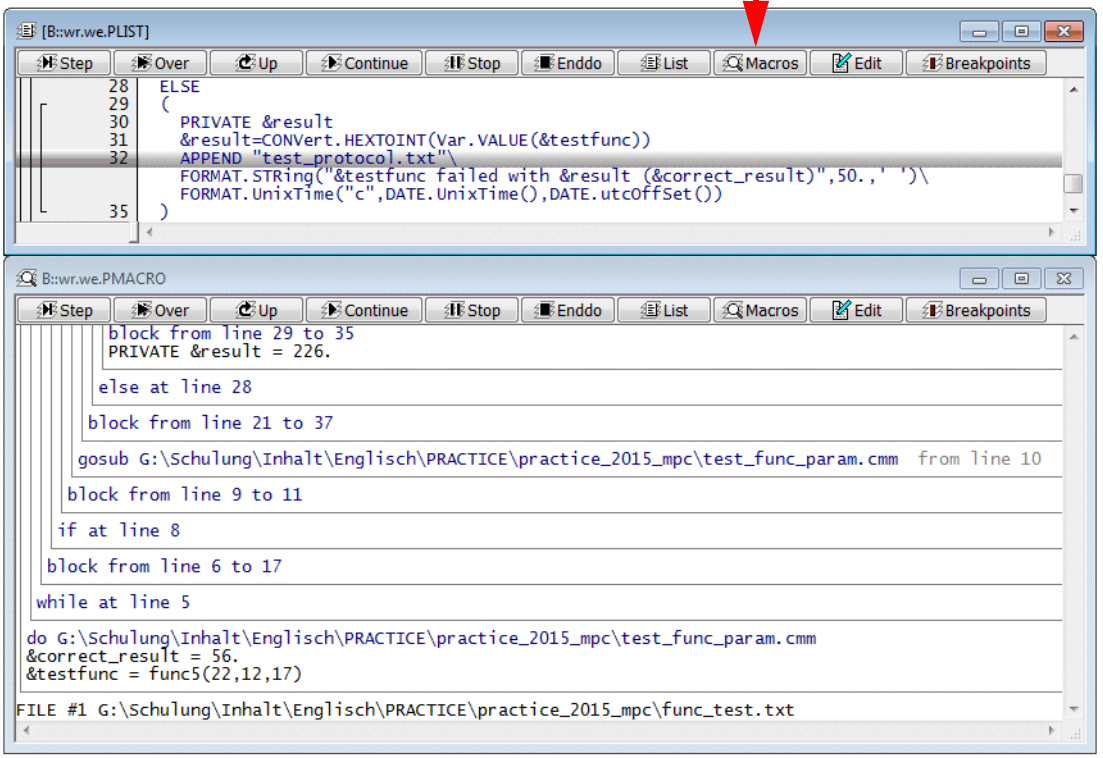
PRACTICE のブレークポイントを設定することができます。

- 指定したスクリプトの特定の行に移動
- スクリプト内の特定の行に移動 (*)



PRACTICE スタックの表示

PRACTICE スタックを表示します。



PRACTICE スタックの表示画面では、プログラムのネスティングと PRACTICE マクロを表示します。

©1989-2020 Lauterbach GmbH

ここまでは、デバッグ時に様々な設定を行うために簡単なスクリプトの記述方法、作成方法を示してきました。以降、PRACTICE が備える高度な言語機能と独自の構文について示します。本セクションでは、言語の構文、プログラム要素、機能、関数の定義などについて示します。PRACTICE スクリプトの例は、~~/demo/practice または、以下の URI を参照してください。

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

プログラム要素

コメント

コメントは // または ; で始まり、次の改行まで継続します。

ただし、PRACTICE の文法では、";" は式の終了指定になります。そのため、";" は Var コマンドグループでは、以下の例に示すように特別な意味を持ちます。

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

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

明示的にコメントを指定する際は、"//" の使用を推奨します。

コマンド

以下の３種類のコマンドが、PRACTICE スクリプトで利用可能です。

- コマンドラインで対話的に利用可能なすべての TRACE32 コマンド
- PRACTICE スクリプトのフロー制御と条件付き実行のためのコマンド
- I/O コマンド

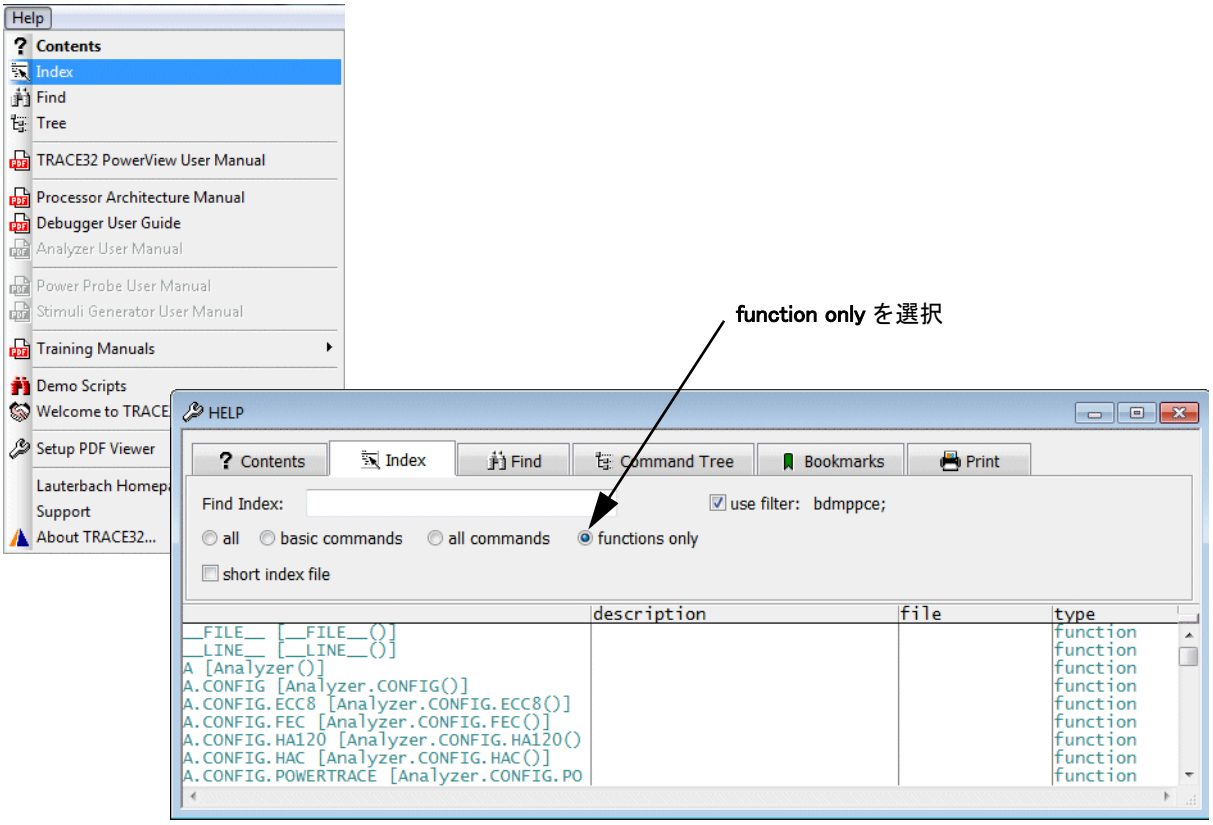
©1989-2020 Lauterbach GmbH

ターゲットシステムや開発ツールの状態に関する情報を取得するための関数です。

Register(<register_name>)	指定した CPU レジスタの値を取得します。
Var.VALUE(<hll_expression>)	HLL 式の内容を取得します。
STATE.RUN()	ターゲット上でプログラムが実行されている場合は、true、実行が停止している場合は false を返します
OS.PresentWorkingDirectory()	現在の作業ディレクトリ名を文字列で返します。
CONVert.CHAR(<value>)	整数値を ASCII 文字に変換します。

©1989-2020 Lauterbach GmbH

利用可能なすべての関数のリストは、オンラインヘルプに記載されています。



ラベル

ラベルは、行の先頭から始まり、":"(セミコロン) で終了していなければなりません。ラベルについての詳細は、サブルーチンのセクションを参照してください。



ラベル文字列内のスペース " " は、PRACTICE の構文解析時に誤認識を発生させることがあります。スペースをラベル文字列内に含めることは推奨されません。

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

&i=7.
```

©1989-2020 Lauterbach GmbH

条件実行

コマンドリスト

IF *〈条件〉*
(
〈if ブロック〉
)
ELSE
(
〈else ブロック〉
)

*〈条件〉*が真ならば *〈if ブロック〉*を実行します。それ以外の場合は *〈else ブロック〉*を実行します。

*〈条件〉*は、TRACE32 の構文で指定します。

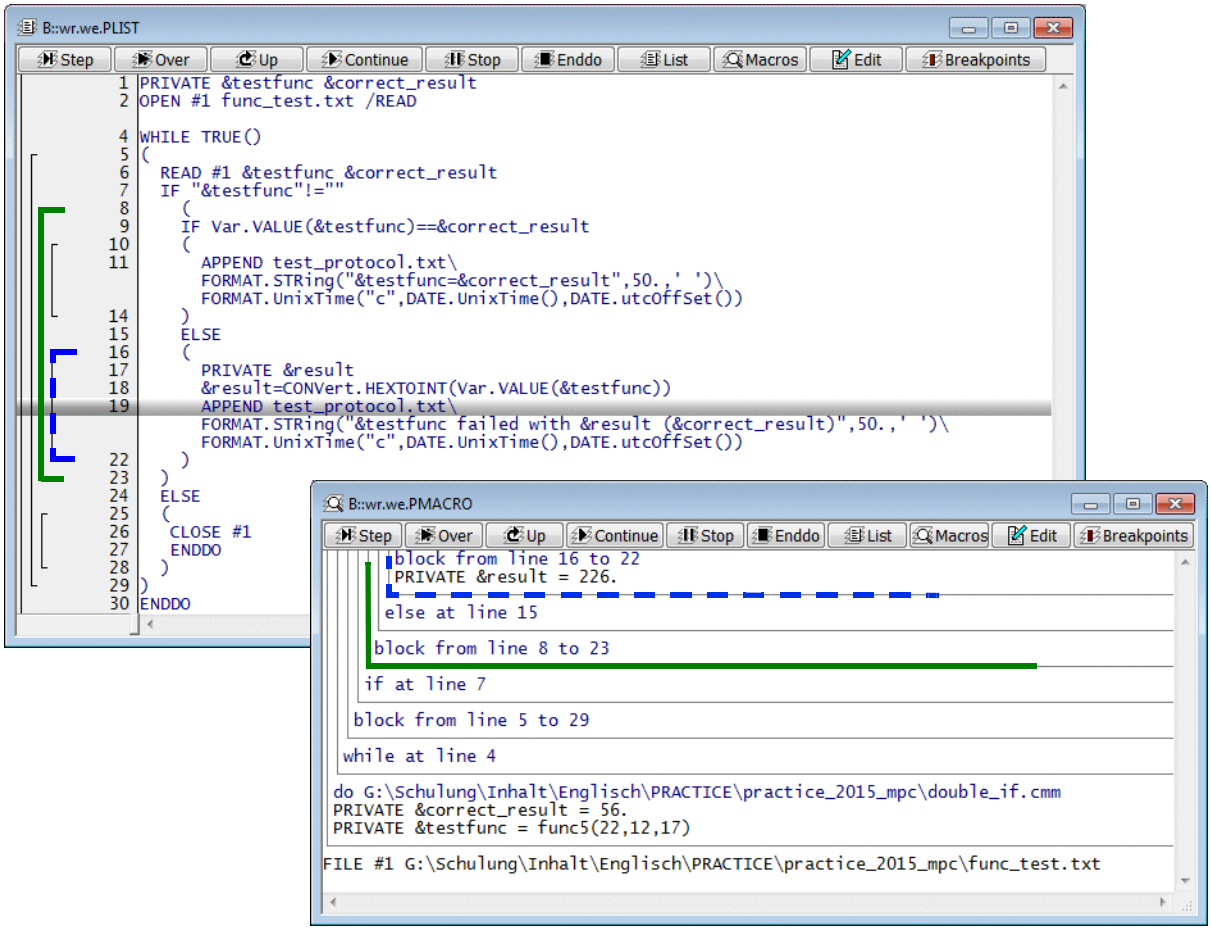
```
// Script double_if.cmm

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

WHILE TRUE()
(
  READ #1 &testfunc &correct_result
  IF "&testfunc"!=""
  (
    IF Var.VALUE(&testfunc)==&correct_result
    (
      APPEND "test_protocol.txt"\
      FORMAT.STRing("&testfunc=&correct_result",50.,' ')\
      FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
    )
    ELSE
    (
      PRIVATE &result
      &result=CONVert.HEXTOINT(Var.VALUE(&testfunc))
      APPEND "test_protocol.txt"\
      FORMAT.STRing("&testfunc failed with &result (&correct_result)",50.,' ')\
      FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffSet())
    )
  )
  ELSE
  (
   CLOSE #1
   ENDDO
  )
)
ENDDO
```

©1989-2020 Lauterbach GmbH

スクリプトの行が長すぎる場合は、スペースに続いて行末に "¥" を付与することで分割して記入することができます。

TRACE32 上の PRACTICE スタックのハンドリング方法は、PRACTICE マクロ (後述) の動作に影響します。スタックのふるまいを理解することで、マクロの動作の理解に役立ちます。

スクリプト内で丸括弧によりブロックが構成されている場合、TRACE32 は、常に PRACTICE スタックに新しいブロックフレームを追加します。



PLIST	現在実行中の PRACTICE スクリプトを一覧表示します。
PMACRO.list	PRACTICE スタックの一覧を表示します。

©1989-2020 Lauterbach GmbH

Var.IF *<hll 条件>*
(
<if ブロック>
)
ELSE
(
<if ブロック>
)

*<hll 条件>*が真ならば *<if ブロック>*を実行します。それ以外の場合は、*<else ブロック>*を実行します。

*<hll 条件>*は、使用するプログラミング言語の構文で記述します。

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

WHILE *<条件>*
(
<ブロック>
)

*<条件>*が真である間、*<ブロック>*を実行します。

*<条件>*は TRACE32 構文で記述します。

Var.WHILE *<hll 条件>*
(
<ブロック>
)

*<hll 条件>*が true の間 *<ブロック>*を実行します。

*<hll 条件>*は、使用するプログラミング言語の構文で記述します。

©1989-2020 Lauterbach GmbH

RePeaT *<カウント><コマンド>*	*<コマンド>* を *<カウント>* 回、繰り返し実行します。
RePeaT *<カウント>* **(** *<ブロック>* **)**	*<ブロック>* を *<カウント>* 回、繰り返し実行します。
RePeaT **(** *<ブロック>* **)WHILE** *<条件>*	*<条件>* が真である間、*<ブロック>* を繰り返し実行します。

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

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

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

©1989-2020 Lauterbach GmbH

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

©1989-2020 Lauterbach GmbH

コマンドリスト

GOSUB *<ラベル>*[*<パラメータ>*] **…** **ENDDO**	*<ラベル>*で指定されたサブルーチンをパラメータを引数として呼び出します。
<ラベル> **(** *<ブロック>* **RETURN** **)**	*<ブロック>*を実行後、呼び出し元に返ります。

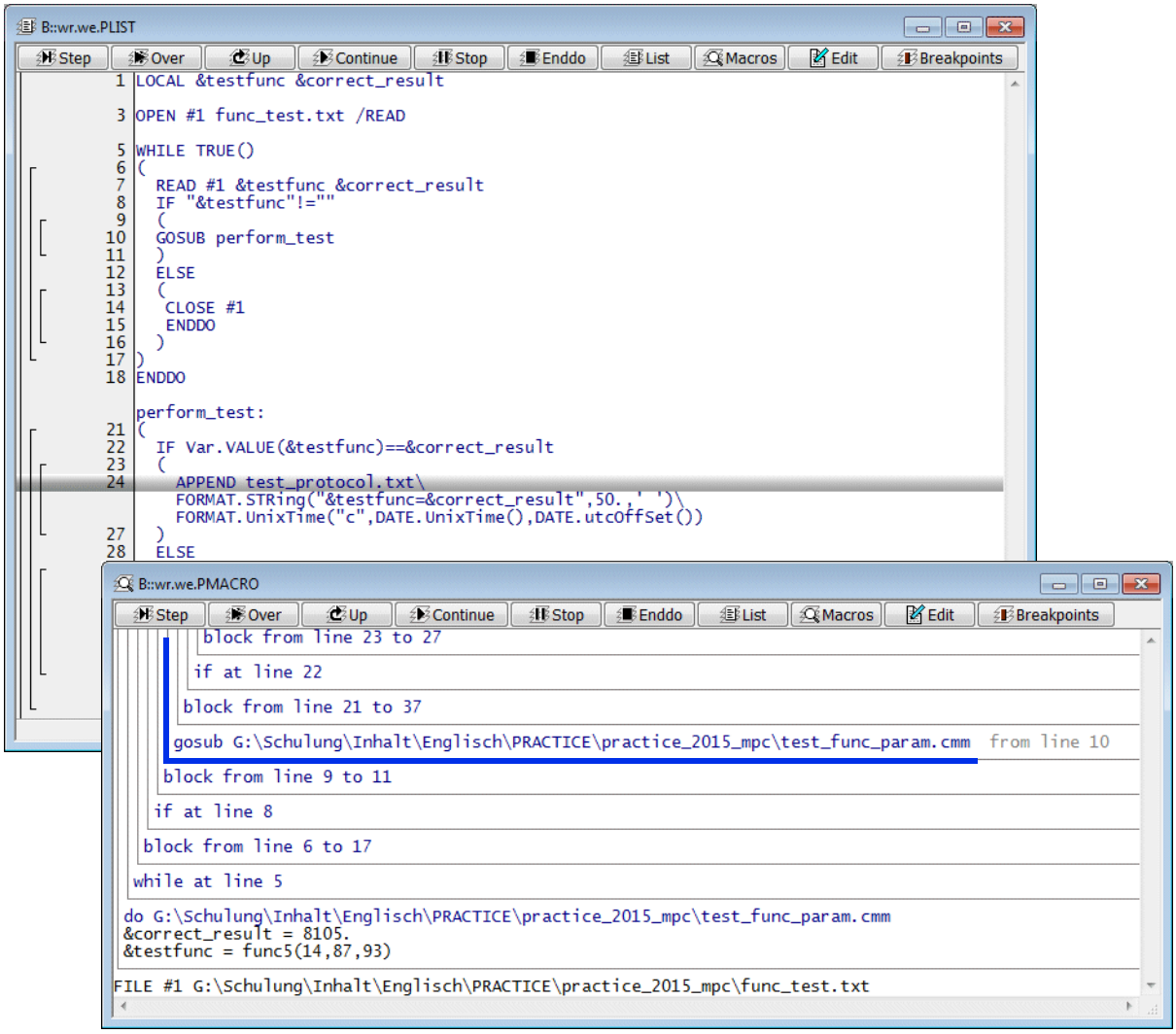
NOTE: ラベルは行の先頭から始まり、":"(セミコロン) で終了していなければなりません。

©1989-2020 Lauterbach GmbH

```
// Script test_func_param.cmm

LOCAL &testfunc &correct_result

OPEN #1 func_test.txt /READ

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

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

©1989-2020 Lauterbach GmbH

TRACE32 は、サブルーチンが呼び出されるたびに、PRACTICE スタックに新しい **gosub-frame** を追加します。



©1989-2020 Lauterbach GmbH

コマンドリスト

GOTO *<ラベル>*	*<ラベル>* まで PRACTICE スクリプトを続行します。 *<ラベル>* は、現在実行中のスクリプトに含まれている必要があります。
JUMPTO *<ラベル>*	*<ラベル>* まで PRACTICE スクリプトを続行します。 *<ラベル>* は、現在実行中のスクリプトに含まれている必要があります。また、<ラベル> はブロック内に存在できません。 PRACTICE のスタックはクリーンされます。

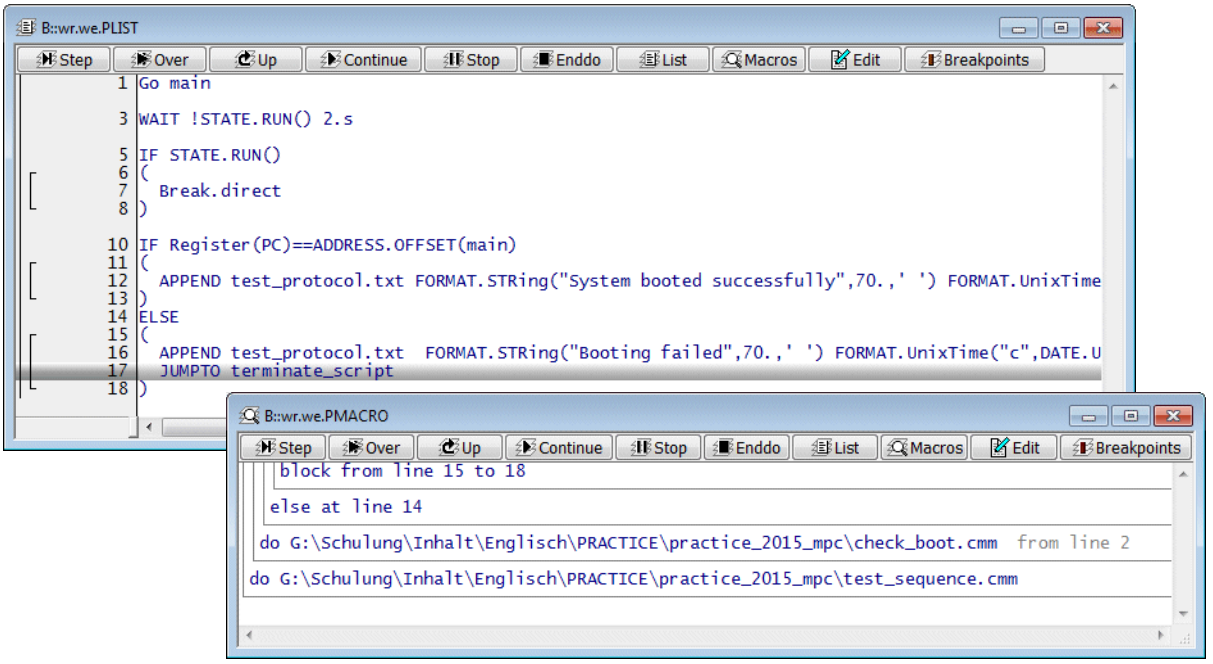
©1989-2020 Lauterbach GmbH

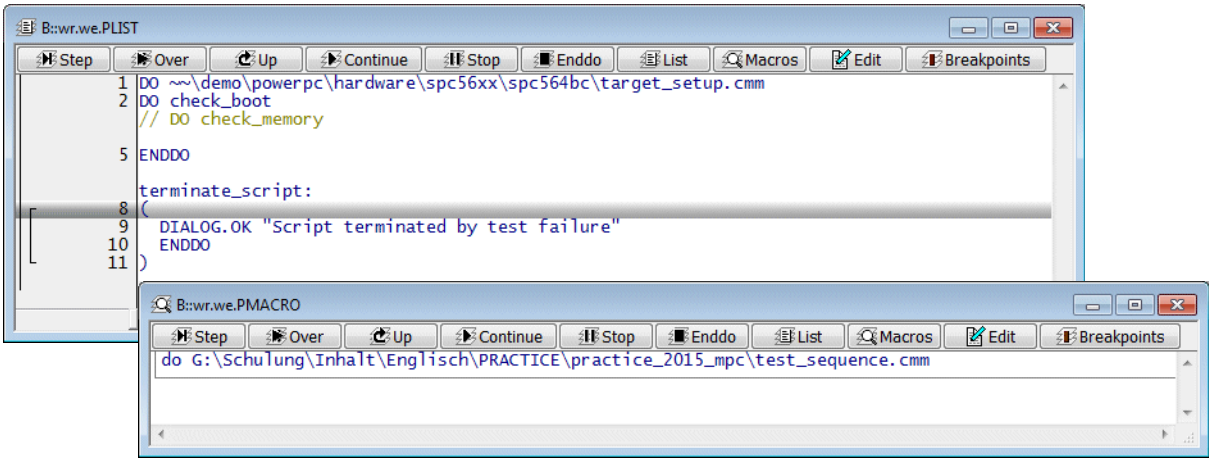
```
// Script test_sequence.cmm

DO target_setup.cmm
DO check_boot.cmm


ENDDO

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

```
// Script check_boot.cmm

Go main

WAIT !STATE.RUN() 2.s

IF STATE.RUN()
(
  Break.direct
)

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

DIALOG.OK *<テキスト>*	*<テキスト>* を伴ってダイアログを表示します。ユーザがダイアログの OK ボタンを押すまでスクリプトの実行を停止します。

©1989-2020 Lauterbach GmbH





©1989-2020 Lauterbach GmbH

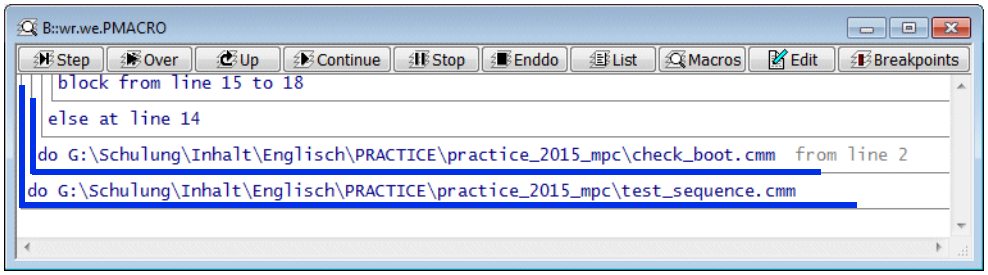
スクリプトの呼び出し

コマンドリスト

DO *<スクリプト>*[*<パラメータ>*]	スクリプトは、コマンドラインから実行、スクリプト内から呼び出しで実行可能です。 引数としてパラメータをスクリプトに渡すことができます。

例

```
// Script test_sequence.cmm

DO target_setup.cmm
DO check_boot.cmm


ENDDO

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



TRACE32 は、スクリプトが起動されるたびに、新しい do-frame を PRACTICE スタックに追加していきます。

©1989-2020 Lauterbach GmbH

PRACTICE マクロ

マクロはスクリプト言語 PRACTICE の変数です。これらは、一連の文字のプレースホルダとして機能します。

PRACTICE のマクロ名は常にアンパサンド記号 ('&') で始まり、その後に文字 (a–z、A–Z)、数字 (0–9)、アンダースコア記号 ('_') を伴います。"&" 後に続く最初の文字は、数字以外を記述します。マクロ名は大文字小文字を区別するため、&a と &A は、異なるマクロ名になります。

マクロの宣言

以下の PRACTICE コマンドのいずれかを使用して、空の PRACTICE マクロをスコープと共に宣言することができます。

PRIVATE *{< マクロ >}*	PRIVATE マクロを宣言します。 PRIVATE マクロは、マクロが宣言されたスクリプト、サブルーチン、ブロック内で機能し、スクリプト、サブルーチン、ブロックの終了時に削除されます。 PRIVATE マクロは、入れ子になったブロックでは機能しますが、呼び出されたサブルーチンや呼び出されたスクリプトでは機能しません。
LOCAL *{< マクロ >}*	LOCAL マクロを宣言します。 LOCAL マクロは、マクロが宣言されたスクリプト、サブルーチン、ブロック内で機能し、スクリプト、サブルーチン、ブロックの終了時に削除されます。 LOCAL マクロは、入れ子になったブロック、呼び出されたサブルーチン、呼び出されたスクリプトで機能します。
GLOBAL *{< マクロ >}*	GLOBAL マクロを宣言します。 GLOBAL マクロは、PMACRO.RESet. コマンドで明示的に削除されるまで、すべてのスクリプト、サブルーチン、ブロックで機能します。スクリプトが実行されていない場合にのみ、GLOCBAL マクロの削除が可能です。

©1989-2020 Lauterbach GmbH

```
// Script test_func_param.cmm

LOCAL &testfunc &correct_result

OPEN #1 "func_test.txt" /READ

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

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

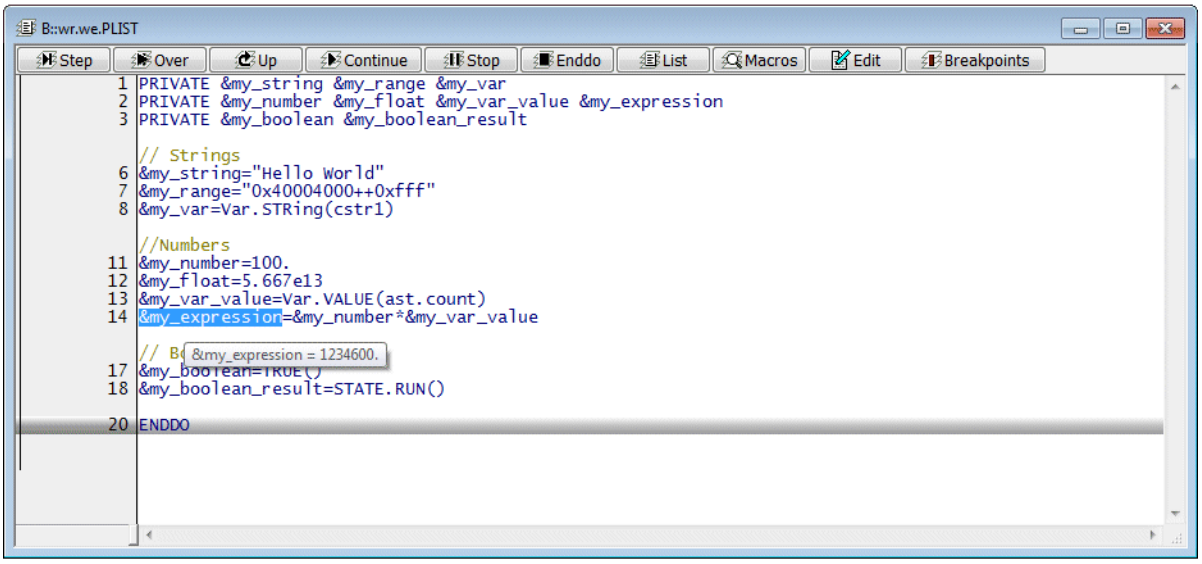
PRACTICE インタープリタは、割り当てが完了すると、暗黙のうちに新しい LOCAL マクロを宣言しますが、そのマクロを現在のスコープ内で見つけることができません。

PMACRO.EXPLICIT コマンドは、PRACTICE マクロが使用されているが、事前に宣言されていない場合は、エラーを生成するよう PRACTICE インタープリタに通知します。同様に、同じ名称のマクロが同一スコープ内で作成された場合も、PRACTICE インタープリタに対してエラーを生成するように通知します。

©1989-2020 Lauterbach GmbH

```
// Script

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

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

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

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

ENDDO
```

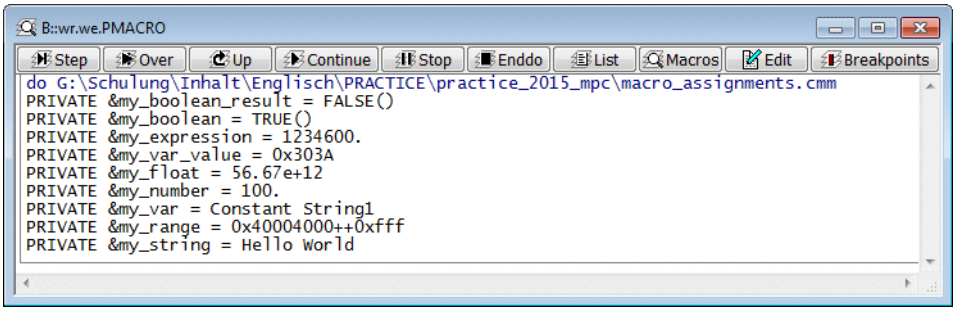
マクロにコンテンツが割り当てられた後、ツールチップで現在のコンテンツを確認することができます。



©1989-2020 Lauterbach GmbH

PRACTICE スタックですべてのマクロの状況が確認できます。

キーワード PRIVATE は、PRIVATE マクロを識別するために使用されます。



Var.STRING(*<hll 表記>*)	*<hll 表記>* が文字または文字配列へのポインタである場合、NULL ターミネートされた文字列が返ります。 上記以外の場合、変数の内容を表す文字列が返ります。

©1989-2020 Lauterbach GmbH

PRACTICE インタプリタがスクリプトを実行する方法を理解することにより、マクロの用法をより深く理解できます。

PRACTICE インタプリタは、スクリプトを１行ずつ実行します。それぞれの行は、以下の３ステップで処理されます。

1. すべてのマクロは、対応する文字列に置き換えられます。
2. すべての式が評価されます。
3. 評価結果によるコマンドが実行されます。

PRINT コマンドを利用した例を示します。

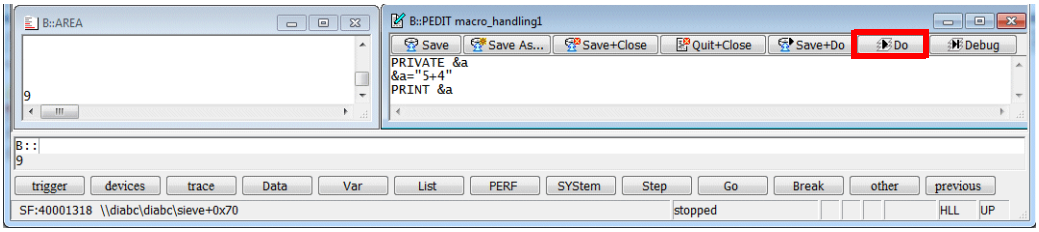
PRINT {<データ>}	指定された *<data>* を TRACE32 メッセージ行と TRACE32 メッセージ領域に **PRINT** します。

例 1

```
PRIVATE &a

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

PRINT &a 行に着目してみましょう。この行を実行するために、インタープリタはまず

1. すべてのマクロを対応する文字列に置き換えます。例えば、以下のようになります。
   PRINT &a -> PRINT 5+4
2. すべての式を評価します。
   PRINT 5+4 -> PRINT 9
3. 結果のコマンド **PRINT 9**. を実行します。

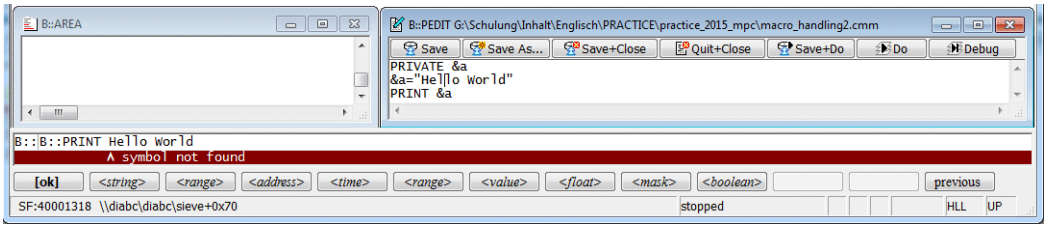


©1989-2020 Lauterbach GmbH

例 2

```
PRIVATE &a

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

この例ではエラーが発生します。



インタープリタが PRINT &a コマンドを実行するため３ステップで解説します。

1. すべてのマクロを対応する文字列に置き換えます。
   PRINT &a -> PRINT Hello World

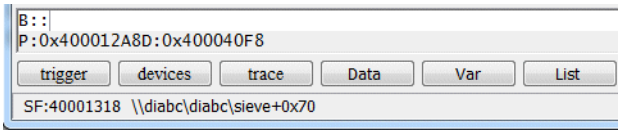
2. すべての式を評価します。
   PRINT Hello World -> **error**

3. コマンドを実行すると、２つ目のステップでエラーが発生し、実行できなくなります。

Hello のような単一の単語（二重引用符で囲まれていない）は、デバッグシンボルとして認識されているため、上記の２つ目のステップでエラーが発生します。

PRACTICE インタープリタがデバッグシンボルを検出した場合、式の評価は、そのデバッグシンボルが参照しているアドレスに置き換えようとします。**Hello** という名称のデバッグシンボルが検出できない場合、PRACTICE インタープリタは、**symbol not found**. というエラーメッセージを出力します。

デバッグシンボル中に **Hello** と **World** が検出された場合、PRINT コマンドは、エラーを発生させずにそれぞれのシンボルのアドレスを表示します。

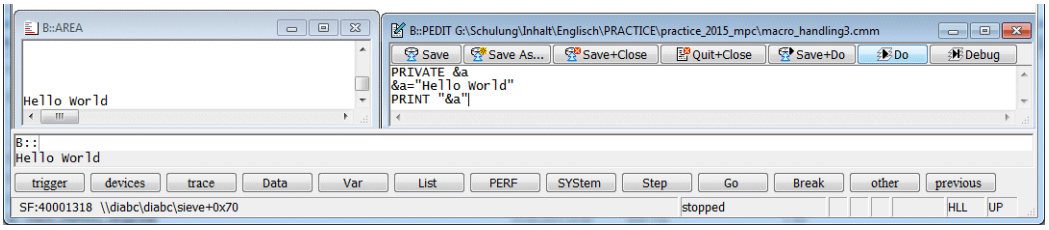


この例では、マクロを文字列で置き換えることが可能であるために、純粋なマクロ置換ステップが基本的に常に動作してしまうことを示していますが、その実行結果は不定となります。

©1989-2020 Lauterbach GmbH

マクロはその文字列で置換可能です。文字列を明示的に文字列として使用したい場合は、マクロを二重引用符で囲む必要があります。

```
PRIVATE &a

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



文字列を明示的に示した実行例として、以下に、**PRINT "&a"** コマンドを利用した実行例を示します。

1. マクロを文字列で置き換えます。
   PRINT "&a" -> PRINT "Hello World"

2. すべての式を評価します。
   この例では何もしません。

3. コマンドを実行します。

©1989-2020 Lauterbach GmbH

文字列の演算の例

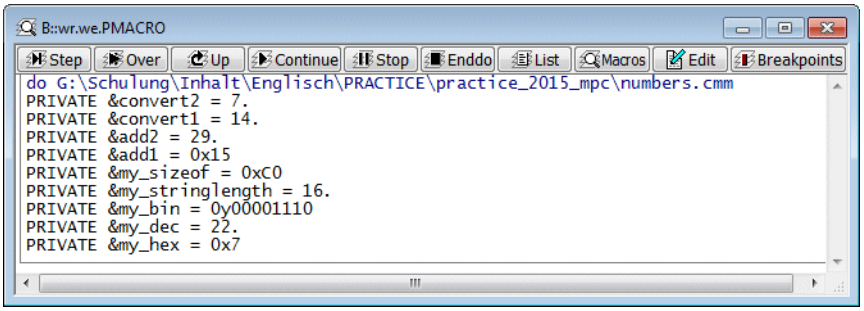
```
// Script string_example.cmm

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

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

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

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

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

DIR *<ディレクトリ>*	指定したディレクトリのファイルとフォルダの一覧を表示します。

©1989-2020 Lauterbach GmbH

```
// Script numbers.cmm

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

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

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

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

// Hex, decimal, binary by CONVERT function

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

PRACTICE スタックには、マクロの値とその基数が表示されます。



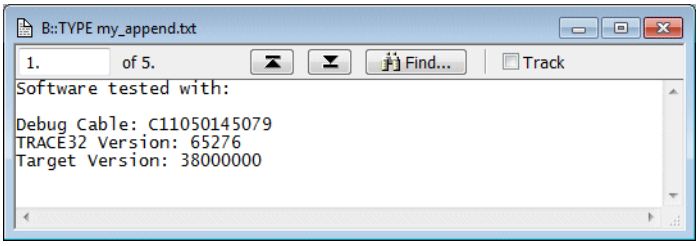
STRing.LENgth(*<文字列>*)	*<文字列>* の長さを１０進数で返します。
sYmbol.SIZEOF(*<シンボル>*)	指定された *<シンボル>*（関数、変数、モジュールなど）が占有するサイズを１６進数で返します。

©1989-2020 Lauterbach GmbH

PRACTICE 出力コマンドを使用した場合、基数情報は出力されません。

```
// Script append_example.cmm

PRIVATE &target_id
&target_id="D:0x40004000"

DEL "my_append.txt"

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



//...
```

上記例の出力結果



TRACE32 コマンド

DEL *<ファイル名>*	*<ファイル名>* で指定したファイルを削除します。

TRACE32 関数

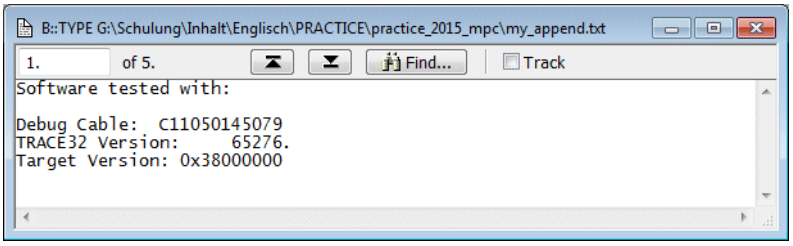
CABLE.SERIAL()	接続されているデバッグケーブルの最初のシリアル番号を返します。
VERSION.BUILD()	TRACE32 ソフトウェアのビルド番号（１０進数）を返します。

©1989-2020 Lauterbach GmbH

１０進数／１６進数を明示的に表示するための例を以下に示します。

```
// Script append_example_format.cmm

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

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

上記の実行結果

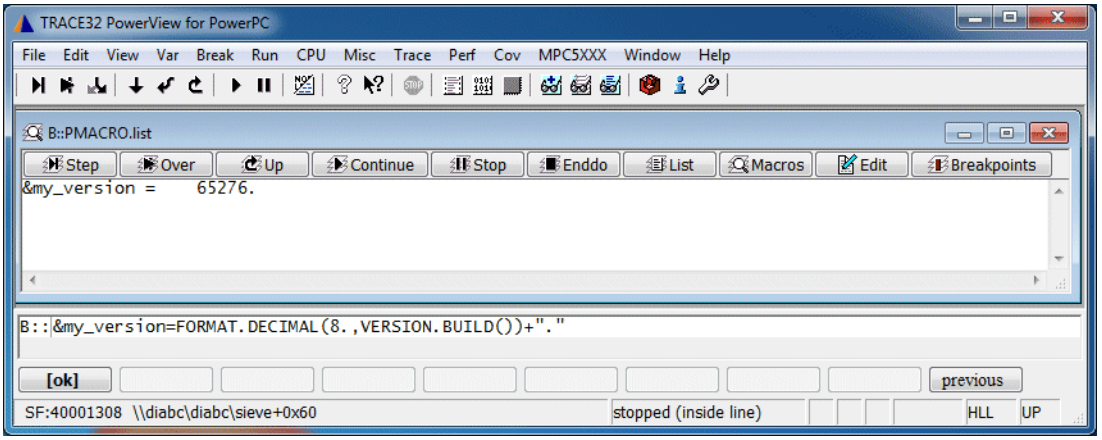


TRACE32 関数

FORMAT.HEX(*<表示桁数>*,*<数値>*)	数値を１６進数にフォーマットし、固定長の *<表示桁数>* と先頭のゼロを含む出力文字列を生成します。

©1989-2020 Lauterbach GmbH

コマンドラインから PRACTICE マクロを直接呼び出すことはできません。また、マクロの利用はスクリプト実行時のみ可能です。マクロの動作確認は、以下のように行います。



©1989-2020 Lauterbach GmbH

すべての複雑なデータ構造に対して、TRACE32 内部変数を利用することが可能です。TRACE32 内部変数宣言は、以下の２つのコマンドを利用します。

Var.NEWGLOBAL *<タイプ>\<変数名>*	グローバルな TRACE32 内部変数を作成します。
Var.NEWLOCAL *<タイプ>\<変数名>*	ローカルの TRACE32 内部変数を作成します。

TRACE32 内部変数は、Data.LOAD コマンドでプログラムをロードする必要があります。ロード後、プログラムから参照されるすべてのデータ型を使用することができます。(sYmbol.List.Type)

- TRACE32 内部変数は PRACTICE マクロと同じスコープを持ちます (PRACTICE スタック上に存在)。
- TRACE32 内部変数は、Var コマンドグループを介して表示、変更されます。

```
; script newlocal.cmm

LOCAL &my_symbol
ENTRY &my_symbol

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

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

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

Var.Assign **%***<フォーマット><値>*	変数の変更により、メッセージ行や AREA ウィンドウにログを生成しません。

sYmbol.TYPE(*<シンボル>*)	シンボルのタイプを数値で返します。 **0** = 存在しないシンボル **1** = 型情報のないプレーンラベル **2** = HLL 関数 **3** = HLL 変数 (上記以外の値は、将来定義される可能性があります。)

©1989-2020 Lauterbach GmbH

プログラムの実行とテストレポートの生成

スクリプト動作概要：プログラムの実行を開始し、関数 main のエントリでプログラムの実行が停止するまで待ちます。

```
// Script run_through_code.cmm

// Part 1

// Prepare debugging
DO "target_setup.cmm"

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

IF STATE.RUN()
(
  Break
)
…
```

本スクリプトの構成

TRACE32 コマンド

Go *<アドレス>*	プログラムの実行を開始します。 プログラムの実行は、*<アドレス>* に到達した時点で停止します。
Break	プログラムの実行を停止します。

©1989-2020 Lauterbach GmbH

DO *<ファイル名>*	*<ファイル名>* で指定された PRACTICE スクリプトを呼び出します。
WAIT *<条件><待ち時間>*	*<条件>* が真になるか、*<待ち時間>* に到達するまで待ちます。

IF *<条件>* **(** *<if ブロック>* **)** **ELSE** **(** *<else ブロック>* **)**	<条件> が真ならば <if ブロック> を実行します。それ以外の場合は <else ブロック> を実行します。 PRACTICE コマンドワード (この場合 IF) の後には、少なくとも 1 つのスペースを記入してください。 <ブロック> は、丸括弧で表記します。PRACTICE では、丸括弧は独立した行に記入する必要があります。

TRACE32 関数

STATE.RUN()	プログラムの実行中は、TRUE を返し、プログラムの実行が停止した場合、FALSE を返します。

©1989-2020 Lauterbach GmbH

スクリプト動作概要：関数 main のエントリでプログラムの実行が停止していることを確認し、テストレポートを生成します。

```
// Part 2
…
IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND "test_protocol.txt" FORMAT.STRing("System booted successfully",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
  ENDDO
)
```

スクリプトの行が長すぎる場合は、スペースに続いて行末に ″¥″ を付与することで分割して記入することができます。

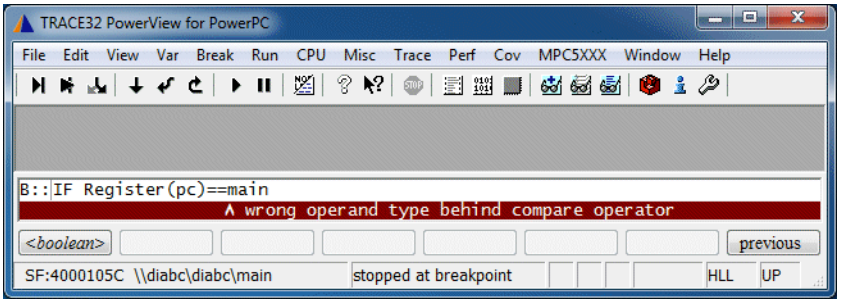
TRACE32 関数

Register(*<レジスタ名>*)	指定したコアレジスタの値を１６進数で返します。

TRACE32 でのアドレスの扱い

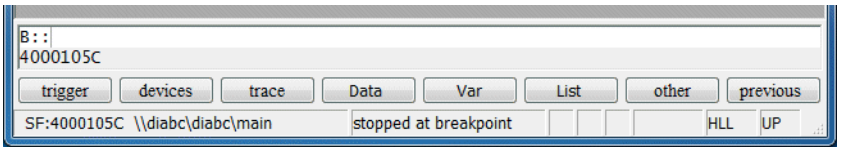
以下のような機能が有用な理由

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

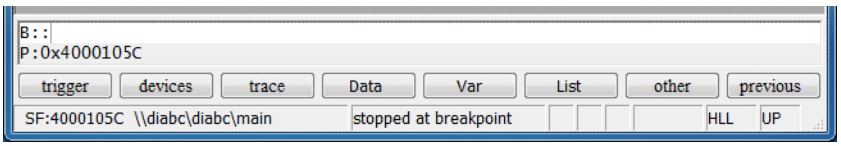


©1989-2020 Lauterbach GmbH

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



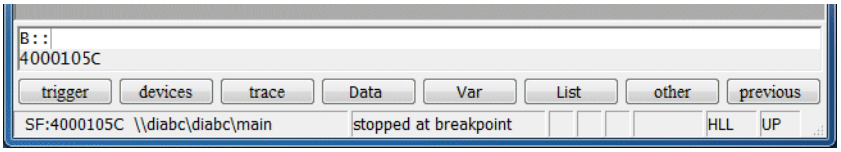

```
PRINT main                              // print the address of main
```



main はアドレスです。PowerView では、アドレスは以下のように表記されます。

- アクセスクラス（この例では ″P:″ は、１文字以上の文字または、数字とコロン ″:″ から構成されます。）
- 実際のアドレス値を示す１６進数の値（この例では 0x4000105C）

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



ADDRESS.OFFSET(*<シンボル>*)　　アドレスを１６進数で返します。

©1989-2020 Lauterbach GmbH

```
// Part 2
…
IF Register(PC)==ADDRESS.OFFSET(main)
(
  APPEND "test_protocol.txt" FORMAT.STRing("System booted successfully",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
)
ELSE
(
  APPEND "test_protocol.txt"  FORMAT.STRing("Booting failed",70.,' ')\
  FORMAT.UnixTime("c",DATE.UnixTime(),0)
  ENDDO
)
```

PRACTICE コマンド :

APPEND *<ファイル名>* {*<データ>*}	ファイル *<ファイル名>* の内容にデータを追加します。
ENDDO	スクリプトは、最後のコマンドもしくは、ENDDO コマンドで終了します。

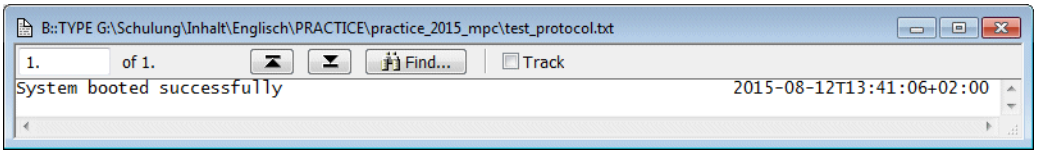
TRACE32 関数 :

FORMAT.STRing(*<文字列>*,*<桁数>*,*<補完文字>*)	*<文字列>* を指定した *<桁数>* に整形します。 *<文字列>* の長さが *<桁数>* より短い場合は、*<補完文字>* が付加されます。
DATE.UnixTime()	１９７０年１月１１日からの経過時間を秒単位、UNIX 形式で返します。
FORMAT.UnixTime("c",DATE.UnixTime(),DATE.utcOffset())	ISO 8601 に準拠した Unix 時間のフォーマットを指定します。

©1989-2020 Lauterbach GmbH

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

Go main

WAIT !STATE.RUN() 2.s

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

上記の実行結果



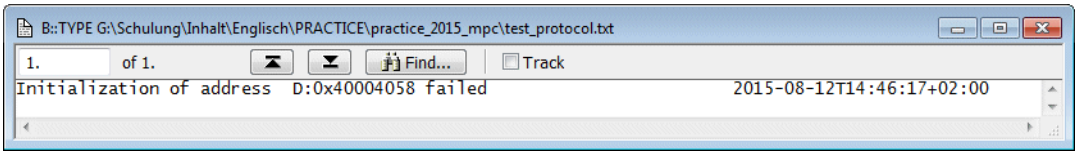
©1989-2020 Lauterbach GmbH

アドレスの内容を確認

スクリプト動作概要：指定したアドレスまでプログラム実行が行われた後、特定のメモリアドレスが正しい値で初期化されているかを確認します。

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

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

上記の実行結果



TRACE32 関数

Data.Long(*<アドレス>*)	指定したアドレスの値を３２ビットの１６進数で返します。

*<アドレス>*は、アクセスクラスを必要とします。

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

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

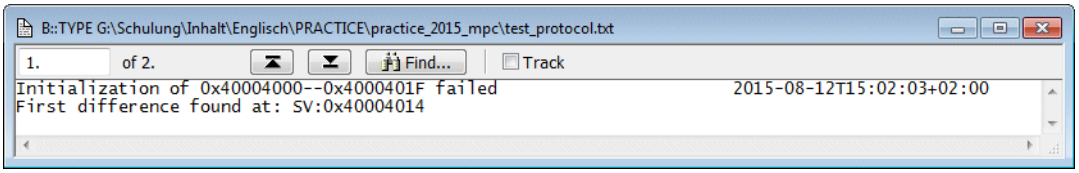
©1989-2020 Lauterbach GmbH

アドレス範囲の内容を確認

スクリプト動作概要：指定したアドレスまでプログラム実行が行われた後、特定のメモリ範囲が正しい値で初期化されているかを確認します。正解値はバイナリファイルを指定しています。

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

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

上記の実行結果



TRACE32 コマンド

Data.LOAD.Binary *<ファイル名><アドレス>* **/DIFF**	TRACE32 関数の FOUND、TRACK.ADDRESS を利用して、<アドレス> のメモリ内容と <ファイル名> の内容を比較し、結果を示します。

TRACE32 関数

FOUND()	違いが見つかった場合に TRUE を返します。
TRACK.ADDRESS()	最初に見つけた差分のアドレスを返します。

©1989-2020 Lauterbach GmbH

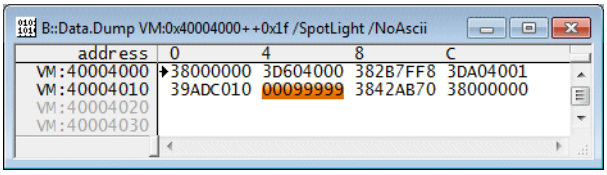
PRIVATE {< *マクロ名* >}	PRIVATE マクロを宣言します。 PRACTICE マクロ名は、デバッグ中のプログラムの変数名と判別するために、& で始めます。 プライベート PRACTICE マクロは、宣言ブロック内でのみ表示され、ブロックが終了すると消去されます。 この例では、丸括弧の間の宣言ブロックに複数命令を含んでいます。

©1989-2020 Lauterbach GmbH

以下のスクリプトは、すべてのデータの差異を詳細に検査する例です。

```
// Script check_memory_range_details.cmm

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

上記の実行結果



TRACE32 コマンド

Data.LOAD.Binary *<ファイル名>* **VM:***<アドレス>*	*<ファイル名>*の内容を TRACE32 仮想メモリ上の *<アドレス>* にロードします。TRACE32 仮想メモリは、実際にはホストコンピュータ上に確保された領域で、実ターゲットメモリと同様にコマンドでの表示 / 変更を行えます。
Data.dump **VM:***<アドレス範囲>* **/SpotLight**	指定された *<アドレス範囲>* の TRACE32 仮想メモリの内容を表示します。 **SpotLight** オプションは、PowerView にメモリ内容が表示されている際に、変更されたメモリ位置をハイライト表示します。
Data.COPY *<アドレス範囲>* **VM:***<アドレス>*	<アドレス範囲> の内容を TRACE32 仮想メモリにコピーします。

PRACTICE コマンド

PRACTICE スクリプトを実行後、**PRINT** コマンドの実行後のみ画面を更新します。

SCREEN.display	画面の即時更新を指示します。

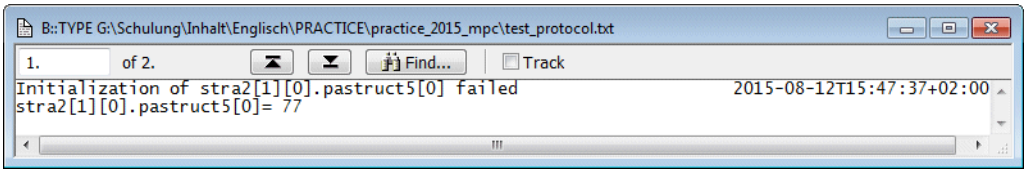
©1989-2020 Lauterbach GmbH

変数の内容の確認

スクリプト動作概要：指定したアドレスまでプログラム実行が行われた後、特定の変数が正しい値で初期化されているかを確認します。

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

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

上記の実行結果



PRACTICE コマンド

Var.IF *〈hll 条件〉* **(** *〈ブロック〉* **)**	使用するプログラミング言語の構文で書かれた *〈hll 条件〉* が真の場合に *〈ブロック〉* を実行します。(この例では、C または , C++ 表記です。)

PRACTICE 関数

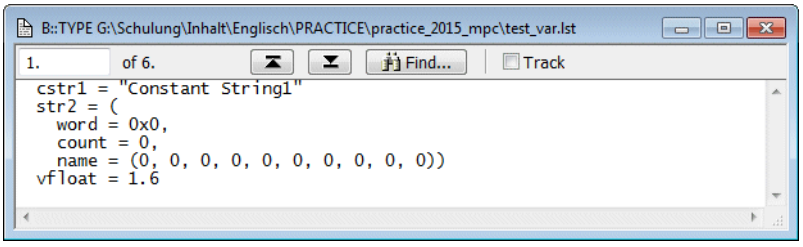
Var.VALUE(*〈hll 表記〉*)	*〈hll 表記〉* で指定した変数の内容を１６進数で返します。

©1989-2020 Lauterbach GmbH

書式を指定して変数を記録

スクリプト動作概要：各種変数の内容をファイルに書き込みます。Var.View コマンドと同じ書式を使用します。

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

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

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

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

PRinTer.CLOSE
…
```



TRACE32 コマンド

PRinTer.FileType *<フォーマット>*	出力ファイルの出力 *<フォーマット>* を指定します。
PRinTer.Open *<ファイル名>*	出力用のファイル *<ファイル名>* をオープンします。
PRinTer.CLOSE	開いているファイルをクローズします。
WinPrint.*<コマンド>*	*<コマンド>* の出力を指定したファイルにリダイレクトします。
WinPos ,,,,,0	デフォルトでは、TRACE32 *<コマンド>* とその出力は、指定されたファイルにリダイレクトされます。 この **WinPOS** コマンドにより、*<コマンド>* の出力のみが指定されたファイルにリダイレクトされます。

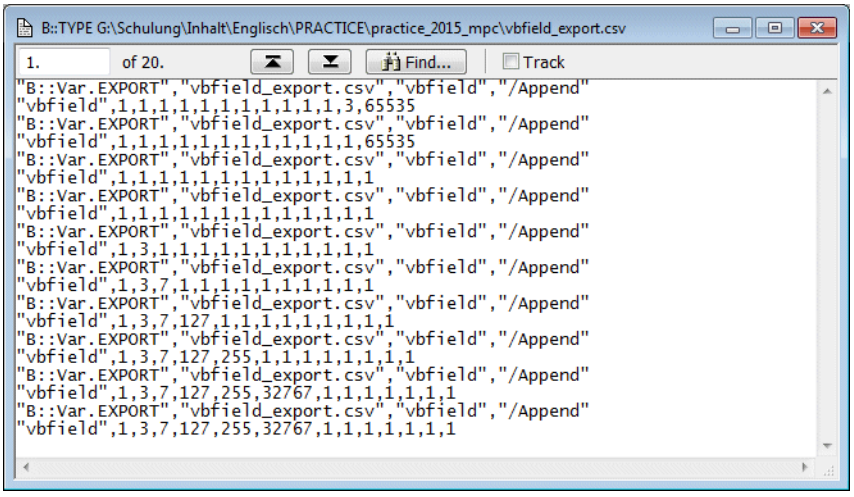
©1989-2020 Lauterbach GmbH

変数を CSV ファイルに記録

スクリプト動作概要：指定したブレークポイントでプログラムの実行が停止するたびに、変数 **vbfield** の内容を CSV ファイルに書き込みます。

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

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

上記の実行結果



©1989-2020 Lauterbach GmbH

// データライトブレークポイントの指定は、デバッグ中の言語 (C, C++, ...) 式が利用可能です。
Var.Break.Set *<hll 表記>* /Write

// データライトブレークポイントの数には限りがあるため、
// 一旦ブレークポイントの設定をリセットすることを推奨します。
Break.RESet

// 変数の内容を CSV(カンマ区切り値) ファイル *<ファイル名>* に追加します。
Var.EXPORT *<ファイル名>* [%*<フォーマット>*] {*<変数名>*} /Append

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

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

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

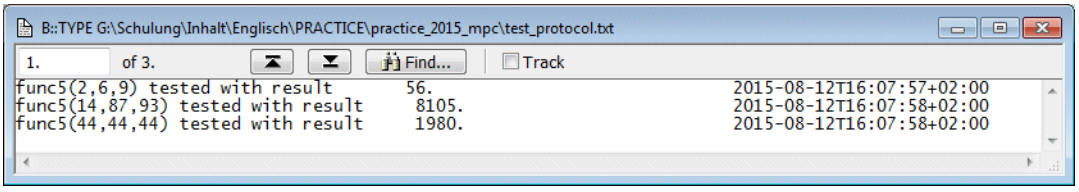
©1989-2020 Lauterbach GmbH

関数の機能テスト

スクリプト動作概要：指定されたパラメータで関数をテストし、テストプロトコルを生成します。

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

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

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


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

上記の実行結果



PRACTICE 関数

FORMAT.Decimal(*<桁数>*,*<値>*)	指定された *<桁数>* にフォーマット（桁数に満たない場合は、スペースを挿入）された１０進表記の数値文字列を生成します。 *<桁数>* よりも多くの文字を必要とする数値の場合は、すべての数値を省かずに文字列を生成します。
Var.VALUE(*<関数コール>*)	呼び出した関数の戻り値を１６進数で返します。

©1989-2020 Lauterbach GmbH

パラメータファイルを利用した関数の機能テスト

スクリプト動作概要：関数の機能テストを、関数名、パラメータ、期待値などをパラメータファイルで定義して実施します。

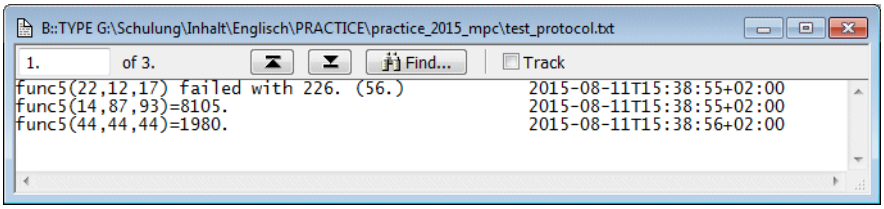
```
// Script test_func_param.cmm

LOCAL &testfunc &correct_result

OPEN #1 "func_test.txt" /READ

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

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

パラメータファイルの例

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

©1989-2020 Lauterbach GmbH

上記の実行結果



PRACTICE コマンド

GOSUB *<ラベル>*	<ラベル> で識別されるサブルーチンを呼び出します。 ラベルは、行の先頭から始まり、":"(セミコロン) で終了していなければなりません。ラベル前に空白を入れてはいけません。 サブルーチンは通常、enddo 文の後に配置されます。
RETURN	サブルーチンから戻ります。
LOCAL *<マクロ>*	LOCAL マクロを宣言します。 LOCAL マクロは、入れ子になったブロック、呼び出されたサブルーチン、呼び出されたスクリプトで機能します。 LOCAL マクロは、宣言ブロックが終了すると削除されます。
WHILE *<条件>* (*<ブロック>*)	*<条件>* が真である限り、*<ブロック>* を実行します。

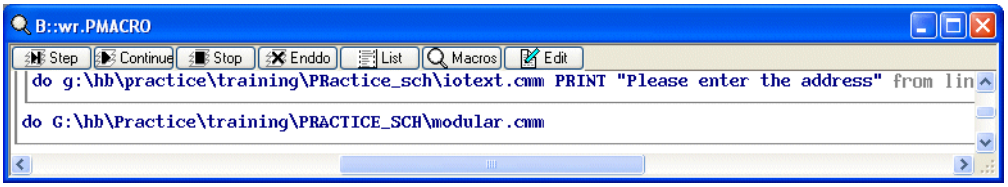
OPEN #*<バッファ番号>* *<ファイル名>* **/Read**	ファイル *<ファイル名>* を読み込み用にオープンします。ファイルは以降のコマンドで *#<バッファ番号>* で参照されます。
READ #*<バッファ番号>* {*<マクロ>*}	*#<バッファ番号>* で参照されたファイルから、1 行を PRACTICE マクロに読み込みます。 スペース ' ' は、パラメータの区切り文字として機能します。
CLOSE #*<バッファ番号>*	*#<バッファ番号>* で参照されたファイルをクローズします。

TRACE32 関数

CONVert.HEXTOINT(*<数値>*)	*<数値>* を 10 進数に変換します。

©1989-2020 Lauterbach GmbH

PRACTICE スクリプト / 関数へのパラメータの受け渡し



ENTRY *<パラメータリスト>*

ENTRY コマンドは以下のように利用できます。

- パラメータを PRACTICE スクリプトまたはサブルーチンに渡します。
- サブルーチンからリターン値を受け取ります。

©1989-2020 Lauterbach GmbH

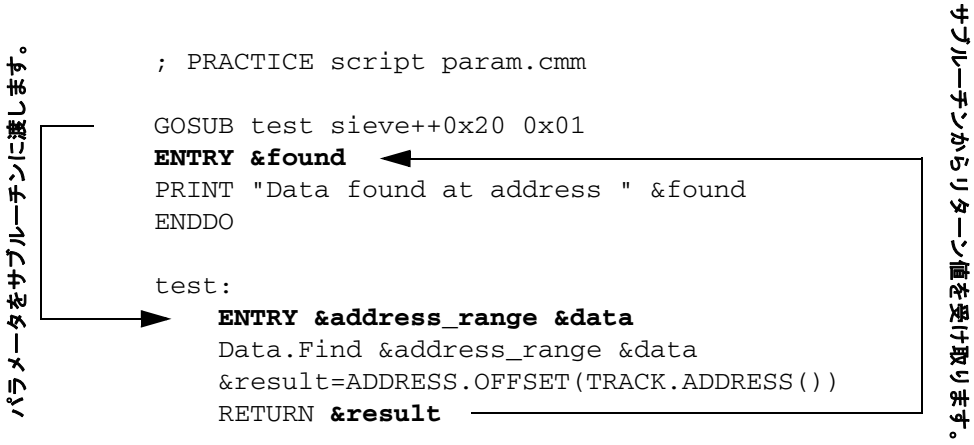
例 1 ：PRACTICE スクリプトにパラメータを渡します。

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

ENTRY &address_range &data_old &data_new

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

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

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

Data.Find &address_range &data_old

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

ENDDO
```

©1989-2020 Lauterbach GmbH

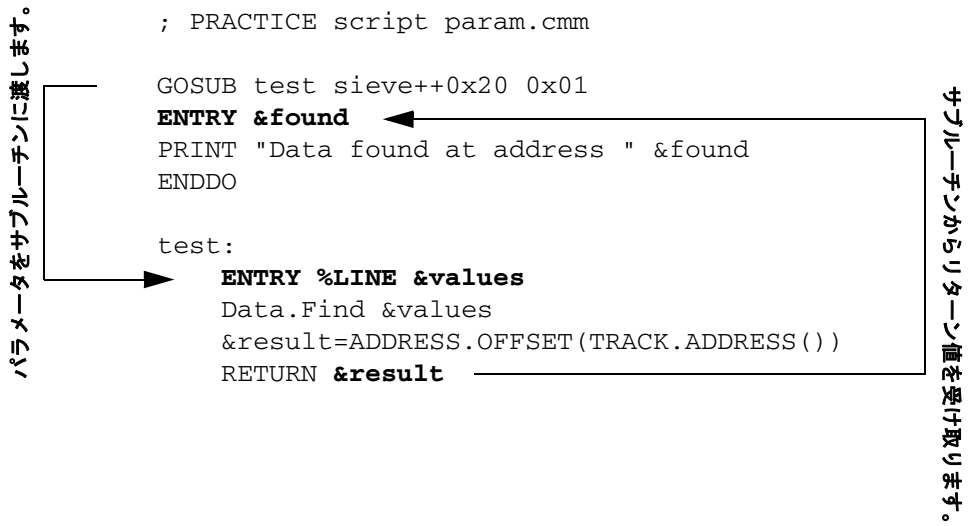
Data.FIND *<アドレス範囲><データ>*	指定した *<アドレス範囲>* 内で *<データ>* を検索します。続いて、TRACE32 関数の **FOUND()**、**TRACK.ADDRESS()** を利用します。
FOUND()	最後の **DATA.FIND** コマンドの実行結果を示すブール値を返します。
TRACK.ADDRESS()	最後に検索に成功した際のアドレスを返します。

例２：サブルーチンにパラメータを渡し、返り値を受け取ります。



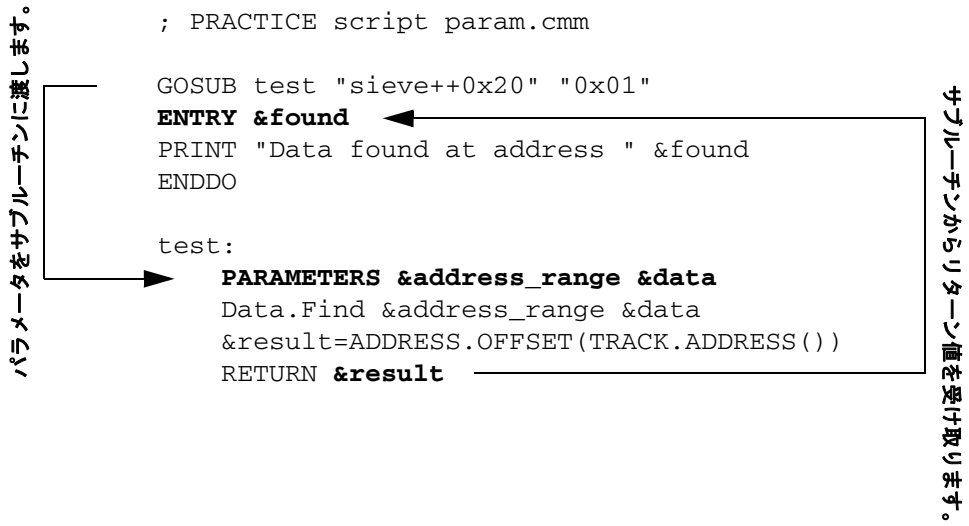
©1989-2020 Lauterbach GmbH

例３：引数を単一のパラメータとしてサブルーチンに渡し、戻り値を取得します。



PARAMETERS 関数は、スクリプトやサブルーチンに渡されたパラメータを読み取ることができます。また、現在のスコープにマクロが存在しない場合、自動的に作成された PRIVATE マクロにパラメータを割り当てます。このときパラメータは引用符(")で囲んで記述する必要があります。

例４：PARAMETERS を利用し引数をサブルーチンに渡し、 戻り値を取得します。



©1989-2020 Lauterbach GmbH

例５：コマンドラインを利用してパラメータを渡す例です。**&data_old** と **&data_new** の値が指定されていない場合、デフォルトの値が代入されます。

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


PARAMETERS &address_range &data_old &data_new

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

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

DATA.Find &address_range &data_old

IF FOUND()
(
   Data.Set TRACK.ADDRESS() &data_new
   Data.Print TRACK.ADDRESS()
   DIALOG.OK "Patch Done"
)
ELSE
(
   DIALOG.OK "Patch Failed."
)
ENDDO
```

スクリプトにパラメータを渡す場合、各パラメータを指定する順番を意識する必要があります。より簡易な方法としては、関数 STRing.SCANAndExtract() を利用する方法があります。

STRing.SCANAndExtract("*<文字列>*","*<検索対象>*","*<デフォルト>*")	*<文字列>* 文字列は、パターン *<検索対象>* を検索し、見つかった場合は、PRACTICE マクロに次の空白までの文字列が代入されます。*<検索対象>* が見つからない場合は、代わりに *<デフォルト>* が代入されます。

©1989-2020 Lauterbach GmbH

.

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

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

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

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

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

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

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

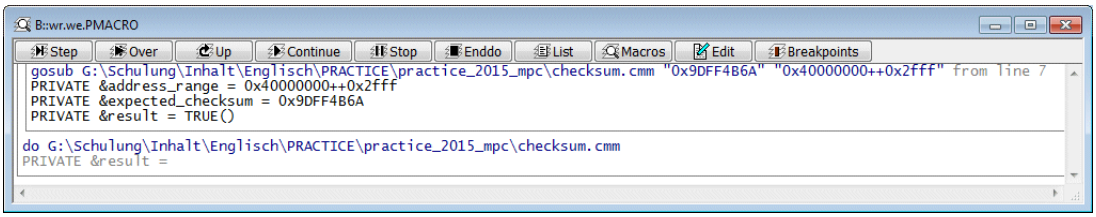
STRing.UPpeR(<*文字列*>**)** <*文字列*>を大文字に変換して返します。

©1989-2020 Lauterbach GmbH

例６：指定された範囲のチェックサムを確認するサブルーチンを呼び出します。.

```
; PRACTICE script checksum.cmm

PRIVATE &result

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

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

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

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

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

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

ENDDO


verify_checksum:
(
   PRIVATE &result
   PARAMETERS &expected_checksum &address_range

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

スコープ外の PRIVATE マクロは、PRACTICE スタックではグレー文字表示になります。



©1989-2020 Lauterbach GmbH

TRACE32 コマンド

AREA.view	TRACE32 メッセージエリアを表示します。
AREA.CLEAR	TRACE32 メッセージエリアをクリアします。
Data.Set *<アドレス><値>*	*<アドレス>* に *<値>* を書き込みます。
Data.SUM *<アドレス範囲>*/CRC32	指定された *<アドレス範囲>* の CRC32 チェックサムを計算します。

PRACTICE コマンド

PRINT %COLOR.*<色>* {*<データ>*}	指定された *<データ>* を指定された *<色>* で TRACE32 メッセージエリアに出力します。

TRACE32 関数

Data.SUM()	Data.SUM コマンドで計算したチェックサムを１６進数で返します。

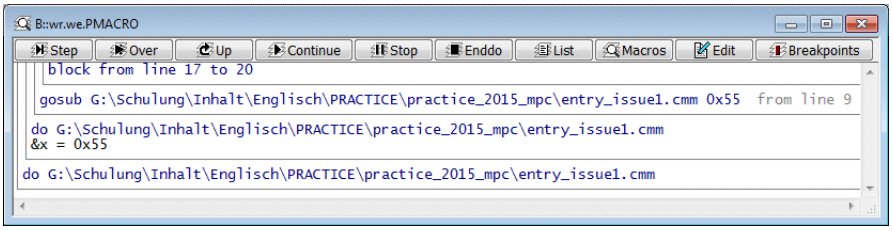
©1989-2020 Lauterbach GmbH

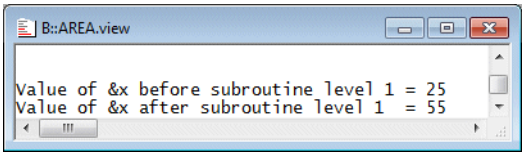
PRACTICE コマンドの PARAMETERS/RETURNVALUES を活用して、ENTRY コマンド利用時に発生する下記のような問題を回避することができます。

1. **LOCAL マクロの意図しない上書き**

   呼び出されたスクリプト / サブルーチン内では、ENTRY コマンドで使用されるマクロが明示的に作成されていない場合、呼び出し元の LOCAL マクロを意図せずに上書きしてしまう可能性があります .

```
// Script entry_issue1.cmm

AREA.view
AREA.CLEAR

LOCAL &x
&x=0x25

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

GOSUB level1 0x55

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

ENDDO


level1:
(
  ENTRY &x
  RETURN
)
```

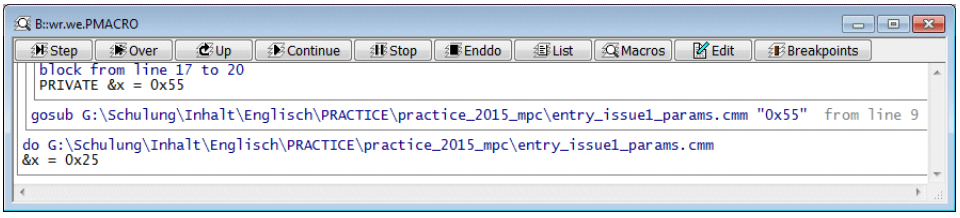


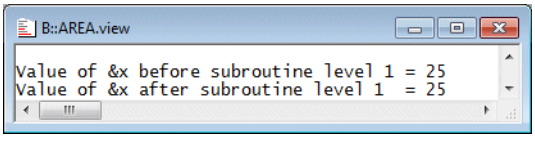


©1989-2020 Lauterbach GmbH

```
// Script entry_issue1_params.cmm

AREA.view
AREA.CLEAR

LOCAL &x
&x=0x25

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

GOSUB level1 "0x55"

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

ENDDO


level1:
(
  PARAMETERS &x
  RETURN
)
```

呼び出し元の LOCAL マクロは PARAMETERS コマンドのスコープ外であることに留意してください。そのため、新しい PRIVATE マクロ &x が作成されています。



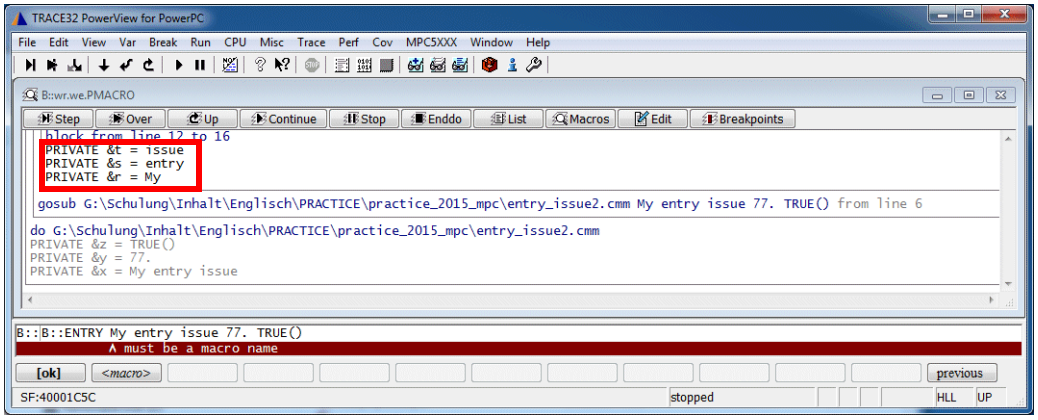


©1989-2020 Lauterbach GmbH

2. パラメータの区切り文字としてスペース ' ' を使用している場合

ENTRY コマンドは、スペースをパラメータの区切り文字とみなします。呼び出されたスクリプト / サブルーチンのパラメータの一つが空白を含む文字列で、かつ引用符で囲まれていない場合、すべてのパラメータに対応したマクロが宣言されていないと認識し、スクリプトはエラーで停止します。

```
// Script entry_issue2.cmm

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

GOSUB level1 &x &y &z

ENDDO


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

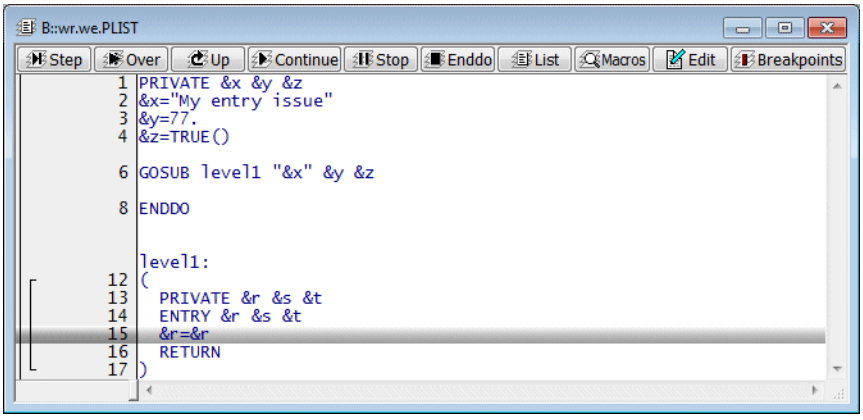


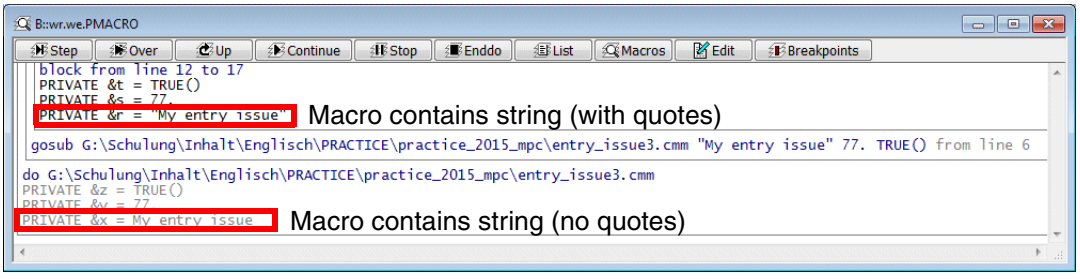
©1989-2020 Lauterbach GmbH

3. 引用符の問題

スペースを含む文字列を引用符で囲み ENTRY コマンドを使用する場合、引用符は文字列の一部として認識されます。

```
// Script entry_issue3.cmm

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

GOSUB level1 "&x" &y &z

ENDDO


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



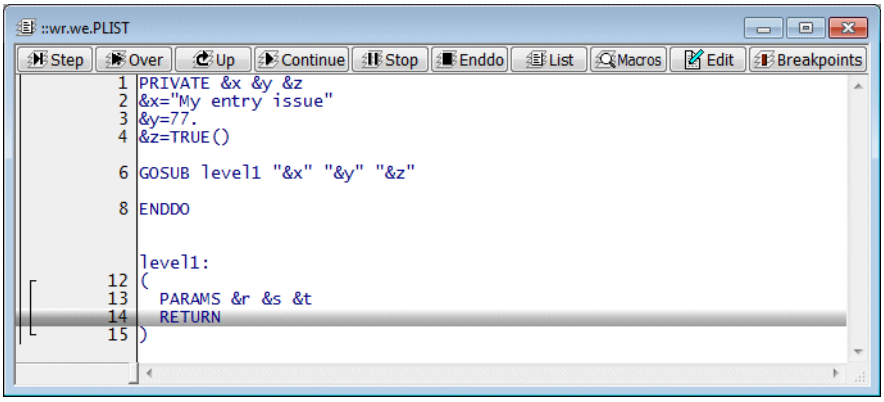


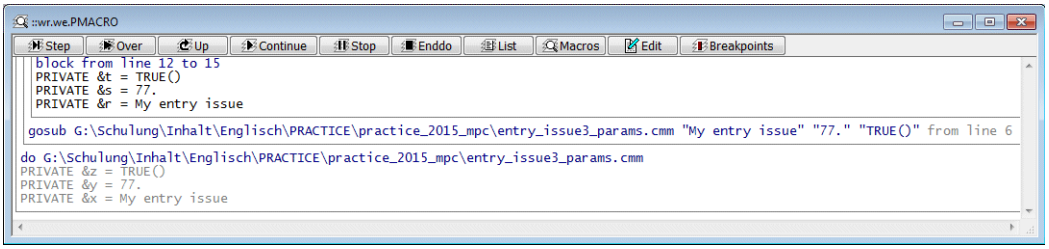

```
// Trick to remove of the quotes
&r=&r
```

©1989-2020 Lauterbach GmbH

PARAMETERS コマンドはスペースまたは、引用符を含む文字列のハンドリングに有用です。

```
// Script entry_issue3_params.cmm

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

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

ENDDO


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





©1989-2020 Lauterbach GmbH

ホストのオペレーティングシステム

PRACTICE スクリプトで、動作するホストコンピュータのオペレーティングシステム (OS) 種別を取得できます。OS を認識することで、動作環境に依存しないスクリプトの記述が可能です。

オペレーティングシステムの検出

OS.VERSION(0) 関数を使用してホストの OS を検出することができます。

```
;Printout host OS
IF OS.VERSION(0)>=0x50
  PRINT "Unkown OS"
ELSE IF OS.VERSION(0)>=0x40
  PRINT "Mac OS X"
ELSE IF OS.VERSION(0)>=0x30
  PRINT "HP-UX"
ELSE IF OS.VERSION(0)>=0x20
  PRINT "Solaris"
ELSE IF OS.VERSION(0)>=0x10
  PRINT "Linux"
ELSE
  PRINT "MS Windows"
ENDDO
```

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

©1989-2020 Lauterbach GmbH

結果の出力

スクリプト実行結果の出力は、ホストのクリップボード、ファイル、接続されたプリンタなどに送ることが可能です。プリンタの設定は、TRACE32 設定ファイル（通常は ~/config.t32）で正しく行われている必要があります。ホストの OS が Windows の場合、多くの場合デフォルトのプリンタ設定で動作します。

結果の印刷

1. **PRinTer.select**（GUI による手動選択の場合）**PRinTer.FILE** または、**PRinTer.ClipBoard** . を利用してプリンタを選択します。
2. **PRinTer.FileType** | **PRinTer.ClipType** / **PRinTer.SIZE** /**PRinTer.OFFSET** を利用して印刷出力をフォーマットします。
3. **PRinTer.OPEN**. を使用してプリンタとの接続を開き、**PRinTer.EXPORT** を使用して CSV データを生成します。
4. **WinPrint**..＜コマンド＞を使用して、コマンドの出力を選択したプリンタにリダイレクトします。
5. **PRinTer.CLOSE** を使用してプリンタ接続をクローズします。

それぞれのコマンドとオプションについての詳細な説明は、PowerView Command Reference (ide_ref.pdf) の "PRinTer" の項を参照してください

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

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

©1989-2020 Lauterbach GmbH

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

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

PRinTer.EXPORT.default "&file"

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

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

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

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

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

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

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

PRinTer.CLOSE

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

©1989-2020 Lauterbach GmbH

環境変数の参照

関数 OS.ENV("< 環境変数名 >") を使用して PRACTICE スクリプトから環境変数を参照可能です。
OS.ENV() から環境変数の値が返り、PRACTICE マクロに代入することができます。.

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

Windows、Linux、MacOS で動作するスクリプトの例が、以下にあります。
~~/demo/practice/my_environment.cmm.

©1989-2020 Lauterbach GmbH

OS コマンドの実行

OS.* ファミリコマンドは、PRACTICE スクリプト内からホスト OS 上のプログラムを起動することができます。ここでは簡単な概要を説明します。より詳しい情報は、PowerView Command Reference (ide_ref.pdf) の "OS" 項を参照してください。

OS.Area *<コマンドライン>*	ホスト OS 上でコマンドを実行します。実行結果は AREA ウィンドウに表示されます。
OS.Command *<コマンドライン>*	ホスト OS 上で複数のコマンドを実行します (ホスト OS 上のパイプ機能を利用)。コマンドラインに PRACTICE マクロを挿入することができます。Windows ホストでは、最初の引数はコマンドのインタプリタ (CMD.EXE など) でなければなりません。 Windows ホストでは、PRACTICE スクリプトはブロックされません。その他のホストでは、'&' を使用しブロックされないよう指定します。
OS.Hidden *<コマンドライン>*	ホスト OS 上でコマンドを実行しますが、その出力は破棄されます。コマンドが完了するまでスクリプトをブロックされます。
OS.screen *<コマンドライン>*	ホスト OS のシェルを開き、コマンドラインを実行します。この動作は、ブロックされません。
OS.Window *<コマンドライン>*	ホスト OS 上でコマンドを実行し、コマンドのすべての出力を TRACE32 ウィンドウにリダイレクトします。これは非インタラクティブであり、コマンドが完了するまで PRACTICE スクリプトはブロックされます。

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

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

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

©1989-2020 Lauterbach GmbH

ファイル操作

PRACTICE スクリプトは、ホスト OS のファイルシステム内のファイルを操作 (表示、コピー、削除など) することができます。スクリプトの移植性を高めるために、OS のネイティブコマンドの代わりにこれらのコマンドを使用することを推奨します。詳細については、PowerView User's Guide (ide_user.pdf) の "File and Folder Operations" を参照してください。

また、PRACTICE スクリプトは、最大９つのファイルハンドル (#1 ～ #9) をファイルのオープン、クローズ、読み取り、書き込み用途で利用可能です。

- ファイルのオープン

  OPEN #*<ファイルハンドル>* *<ファイル名>* / **Read** | **Write** | **Create** | **Binary**

- ファイルのクローズ

  CLOSE #*<ファイルハンドル>*

- オープンしたファイルからデータの読み出し

  READ #*<ファイルハンドル>* [**%LINE**] *<パラメータ>*

- オープンしたファイルへデータの書き出し

  WRITE #*<ファイルハンドル>* *<パラメータ>*

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

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

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

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

©1989-2020 Lauterbach GmbH

バイナリデータは、WRITE または、WRITEB を利用して書き込みます。

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

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

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

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

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

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

©1989-2020 Lauterbach GmbH

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

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

TYPE &file
ENDDO
```

©1989-2020 Lauterbach GmbH

バイナリデータのハンドリングは以前の項で説明したファイルへの出力方法と組み合わせ、ターゲットシステムからの情報の抽出、解析を可能にします。以下の例では、FreeRTOS ベースのシステムで実行中のすべてのスレッドのリストを作成し、整形してダイアログで表示する方法を示しています。

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

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

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

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

DIALOG.END
ENDDO
```

©1989-2020 Lauterbach GmbH

時間と日付

日付と時刻は、それぞれ DATE.DATE() 関数、DATE.TIME() 関数を用いて取得することができます。日付および時刻を扱う関数の詳細は、PowerView Function Reference (ide_func.pdf) の “DATE Functions” を参照してください。

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

これらの関数を利用し、レポートやログファイルにタイムスタンプを追加したり、ログファイル名に日時を含めることができます。

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

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

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

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

PRINT "Creating new log file: &logfile"

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

TYPE "logdir/&logfile"
ENDDO
```

©1989-2020 Lauterbach GmbH

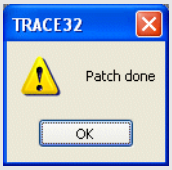
出力コマンド

PRINT *〈フォーマット〉〈パラメータ〉*

DIALOG.OK *〈メッセージ〉*

```
PRINT "FLASH programmed successfully"

PRINT %ERROR "FLASH programming failed"
```

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

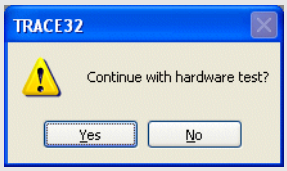


入力コマンド

ENTER	*〈パラメータ〉*	入力
INKEY	[*〈パラメータ〉*]	入力コマンド（文字型）
DIALOG.YESNO	*〈メッセージ〉*	標準ダイアログの生成
DIALOG.File	*〈メッセージ〉*	ダイアログ経由でファイル名の取得

©1989-2020 Lauterbach GmbH

```
INKEY                                                   ; Wait for any key

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

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




```
ENTRY &result

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

```
DIALOG.File *sre
ENTRY &filename

Data.LOAD.S3record &filename

ENDDO
```

©1989-2020 Lauterbach GmbH

AREA ウィンドウ経由の入出力

PRACTICE スクリプトの入出力は、I/O ウィンドウが必要となり、通常 **AREA** ウィンドウ内で処理されます。

AREA ウィンドウの生成、割り当て

1. **AREA** ウィンドウを生成します。

   AREA.Create [*<AREA 名>*]

2. PRACTICE スクリプト入出力用の **AREA** ウィンドウを指定します。

   AREA.Select [*<AREA 名>*]

3. **AREA** ウィンドウの表示位置の指定、命名を行います。このコマンドにより、入出力動作に利用した AREA ウィンドウの後始末などの処理を記述することができます。

   WinPOS[*<ポジション>*] [*<サイズ>*] [*<スケール>*] [*<ウィンドウ名>*] [*<状態>*] [*<ヘッダ>*]

4. **AREA** ウィンドウの表示

   AREA.view [*<AREA 名>*]

AREA ウィンドウの消去

1. **AREA** ウィンドウの設定をデフォルトの設定にリセットします。エラーやシステムメッセージは、メッセージエリア (AREA A000) に表示されます。他の **AREA** ウィンドウを、アクティブすることはありません。

   AREA.RESet

2. 任意のウィンドウの消去

   WinCLEAR [*<ページ名>* | *<ウィンドウ名>* | **TOP**]

©1989-2020 Lauterbach GmbH

```
; PRACTICE file iowindow.cmm

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

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

WAIT 2.s

AREA.RESet
WinCLEAR IO1

ENDDO
```

©1989-2020 Lauterbach GmbH

ON ERROR *<コマンド>*	PRACTICE スクリプトのエラー発生時にコマンドを実行します。
ON SYSUP *<コマンド>*	デバッガと CPU 間の通信が確立した際に < コマンド > を実行します。
ON POWERUP *<コマンド>*	ターゲットの電源投入時に < コマンド > を実行します。

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

```
ON POWERUP GOTO startup

IF !(STATE.POWER())
     STOP

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

ENDDO
```

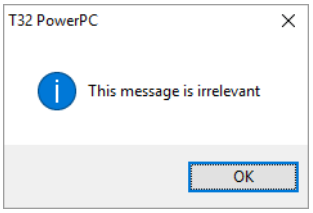
©1989-2020 Lauterbach GmbH

ダイアログ

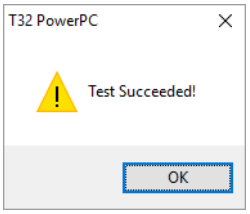
TRACE32 では、ユーザへの通知や、ユーザの選択を促すための、シンプルなダイアログ表示機能をいくつか提供しています。

DIALOG.MESSAGE *"<テキスト>"*	情報アイコンと *<テキスト>* が表示された標準のダイアログボックスを表示します。OK ボタンのクリックによりダイアログは終了します。
DIALOG.OK *"<テキスト>"*	エクスクラメーションマークと *<テキスト>* を含む標準のダイアログボックスを表示します。**OK** ボタンのクリックによりダイアログは終了します。
DIALOG.YESNO *"<テキスト>"*	クエスチョンマークと *<テキスト>* を含む標準のダイアログボックスを作成します。ダイアログ上には **YES**/**NO** の 2 つのボタンが表示され、**YES** のクリックでは真の値が、**NO** のクリックでは偽の値がスクリプトに返されます。**YES**/**NO** どちらのボタンをクリックしてもダイアログを閉じます。

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

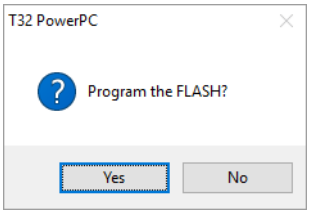



```
DIALOG.OK "Test Succeeded!"
```



©1989-2020 Lauterbach GmbH

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



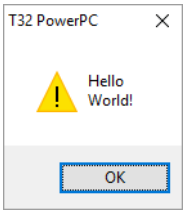
これらのダイアログでは、テキストを複数行に分割して表示することができます。以下の例では複数行に分割する方法を示しています。

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

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

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

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

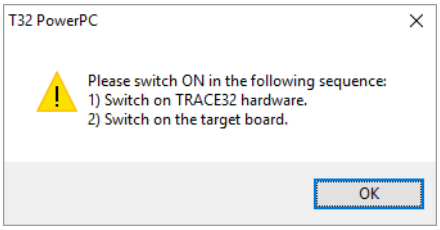


最初３つの例では、キャリッジリターン文字 (0x0D, '¥r') や、改行文字 (0x0A, '¥n') を文字列に挿入するために、CONVert.CHAR() 関数を使用しています。４つ目の例では、二つの文字列を指定し、TRACE32 は一行に一つの文字列を挿入しています。

©1989-2020 Lauterbach GmbH

ダイアログへの複数行の表示を利用することで、より多くの情報表示を分かりやすく提示することが可能です。

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



©1989-2020 Lauterbach GmbH

ダイアログのプログラム

PRACTICE スクリプトでは、複雑なダイアログを生成することができます。これにより、スクリプト用のカスタムインターフェースを表示することができます。ダイアログの生成には、２つの方法があります。最初の方法は、ダイアログのコードを .dlg という拡張子を持つ別のファイルに記述し、DIALOG.view コマンドでファイルを呼び出す方法です。

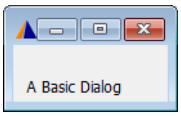
DIALOG.view *<ファイル名>*	ダイアログファイル *<ファイル名>* のコードに基づいてダイアログを作成します。

```
; File basicdialog.dlg

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


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

第２方法は、スクリプト自体の中にダイアログのコードを埋め込む方法です。

```
; Example code to create a simple dialog

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

第１、第２どちらも同じ動作結果になります。ダイアログは、配置されたすべてのコントロールを包含するように自動的にサイズを調整します。



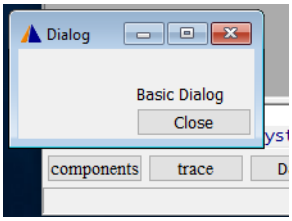
DIALOG.END	現在アクティブなダイアロをクローズします。

©1989-2020 Lauterbach GmbH

非 MDI を指定している場合を除き、デフォルトでのダイアログの表示先は TRACE32 の親ウィンドウ内になります。WinExt コマンドを使うことにより、ダイアログのはフリーフローティングウィンドウとして作成することができます。.

WinExt.<コマンド>	外部ウィンドウを作成するための準備コマンドです。

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



この例で PRACTICE コマンドの STOP コマンドを使用した場合、ダイアログがモーダルになります。スクリプトは、ユーザ (またはスクリプト) が実行を継続するまで待機します。CONTinue コマンドは、BUTTON コントロールに関連付けられたコマンドが実行されたときに発生します。これは、ユーザが "Close" ボタンをクリックした際にアクティブになります。

コマンドのプレフィックス WinResist を追加した場合、WinCLEAR. を呼び出した後もダイアログは表示され続けます。各ダイアログには複数のコマンドプレフィックスを適用することが可能です。

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

©1989-2020 Lauterbach GmbH

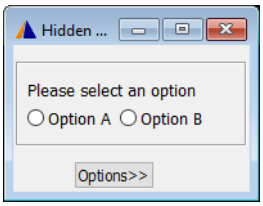
ダイアログのサイズを変更するには WinRESIZE コマンドを使用します。このコマンドは動的に実行することが可能で、非表示オプションも使用可能です。

WinRESIZE [*<幅>*] [*<高さ>*] [*<ウィンドウ名>*]	*<ウィンドウ名>* の開いているウィンドウのサイズを変更します。*<ウィンドウ名>* が指定されていない場合は、一番上位のウィンドウのサイズが変更されます。*<ウィンドウ名>* は WinPOS コマンドまたは NAME コマンドで指定します。

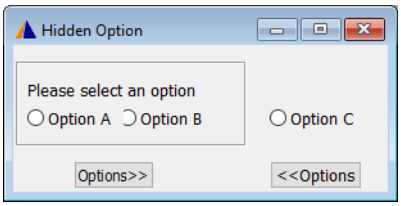
```
WinExt.DIALOG
(
     HEADER "Hidden Option"
     POS 0. 0. 20. 4.
     BOX ""
     POS 1. 1. 15. 1.
     TEXT "Please select an option"
     POS 1. 2. 8. 1.
A.A: CHOOSEBOX "Option A" ""
     POS 9. 2. 8. 1.
A.B: CHOOSEBOX "Option B" ""
     POS 22. 2. 8. 1.
A.C: CHOOSEBOX "Option C" ""
     POS 5.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

©1989-2020 Lauterbach GmbH

A



B



A "Options>>" をクリックするとダイアログを拡張します。

B "<<Options" をクリックすると元のダイアログの表示に戻ります。

サイズ、位置の制御

コントロールの配置はダイアログの左上 (0, 0) を起点とします。ダイアログのデフォルトのサイズは幅 9unit、高さ 1unit です。コントロールが配置された後、次の位置は Y 座標を 1 つ進め、同じ X 座標を維持したまま計算されます。サイズのデフォルトは、最後に配置されたコントロールと同じサイズです。次に配置するコントロールのサイズと位置を上書きするには、**POS** コマンドを使用します。

POS [*<x>*] [*<y>*] [*<幅>*] [*<高さ>*]	ダイアログに配置される次のコントロールのサイズと位置を定義します。

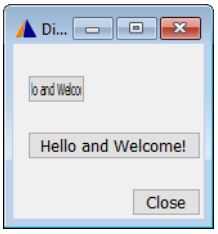
<x>	0 から 16383.5 unit まで、0.5 unit で指定可能です。
<y>	0 から 8191.75 unit まで、0.25 unit で指定可能です。
<幅>	0 から 16383.5 unit まで、0.5 unit で指定可能です。
<高さ>	0 から 8191.75 unit まで、0.25 unit で指定可能です。
,	以前に指定した POS 引数の値が参照されます。ここでの例では、<幅> の以前の値が参照されています。 ; <x> <y> <幅> <高さ> POS 3. 7. , 2.
引数なし	右から順に、直前に実行した POS 引数の値が使用されます。以下の例では、<幅> と <高さ> の値は、直前に実行した POS コマンドから使用されます。 ; <x> <y> <幅> <高さ> POS 3. 7.

上記の例では、**POS.** への引数の数字の最後にピリオド "." を付与しています。ピリオドを付与した数値の記述は、TRACE32 にこの数値を１０進数として扱うように指示するための記述です。

©1989-2020 Lauterbach GmbH

テキストを含むコントロールは、そのサイズをテキストに合わせます。

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



©1989-2020 Lauterbach GmbH

ダイアログのコントロールは、PRACTICE ラベルとの関連付けを必要とするものがあります。このコントロールは、いくつかの追加機能を提供します。

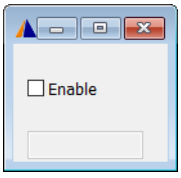
- PRACTICE スクリプトは、コントロールを有効または無効にすることができます。
- PRACTICE スクリプトは、ラベルを使用してコントロールに値を設定することができます。
- PRACTICE スクリプトは、コントロールの状態を設定することができます。
- The PRACTICE script can set the control's state.
- コントロールにコマンドブロックが関連付けられている場合、実行することができます。

©1989-2020 Lauterbach GmbH

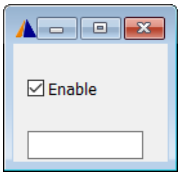
DIALOG.Enable *<ラベル>*	*<ラベル>* に関連付けられたコントロールを有効にします。
DIALOG.Disable *<ラベル>*	*<ラベル>* に関連付けられたコントロールを無効にします。

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

ENDDO
```

A



B



A ダイアログを開くと、CHECKBOX が有効、EDIT コントロールが無効になっています。

B CHECKBOX をクリックすると、EDIT コントロールが有効になります。

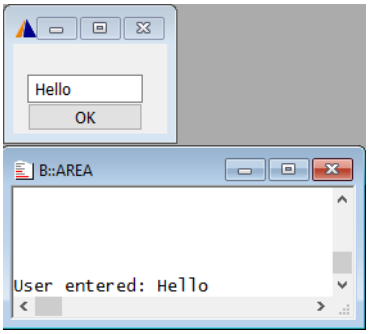
DIALOG.BOOLEAN(*<ラベル>*)	*<ラベル>* に関連付けられたダイアログ内コントロールのブール値を返します。

©1989-2020 Lauterbach GmbH

コントロールからの情報は、そのコントロールに関連付けられたラベルを介して PRACTICE スクリプトで利用できます。DIALOG.STRing() 関数を利用し、コントロールに関連付けられたラベルを取り出します。

DIALOG.STRing(<ラベル>)	<ラベル>に関連付けられたダイアログコントロールのデータを返します。.
DIALOG.STRing2(<ラベル>)	ダイアロ中のコントロールの完全なリスト(LISTBOX, MLISTBOX, DLISTBOX, COMBOBOX など)を取得します。返される値はカンマ区切りの文字列になります。

```
LOCAL &userInput

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

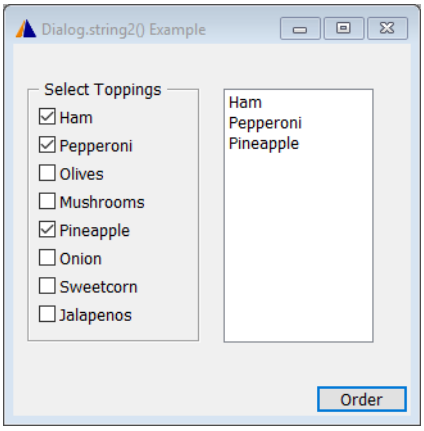


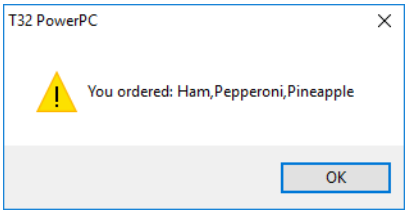
©1989-2020 Lauterbach GmbH

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

©1989-2020 Lauterbach GmbH

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

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

RETURN
```



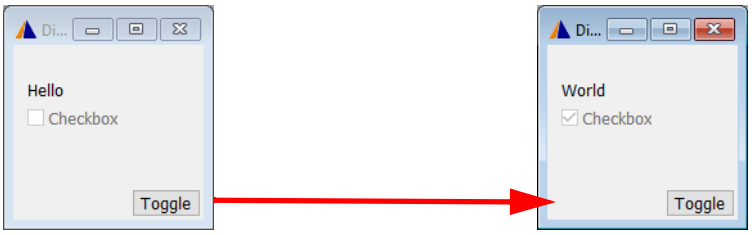


©1989-2020 Lauterbach GmbH

ラベルに関連付けることができるコントロールは、DIALOG.Set コマンドを使用し、データの入力、設定の変更が行えます。

DIALOG.Set *<ラベル><値>*	*<ラベル>* に関連付けられたコントロールに *<値>* を設定します。

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



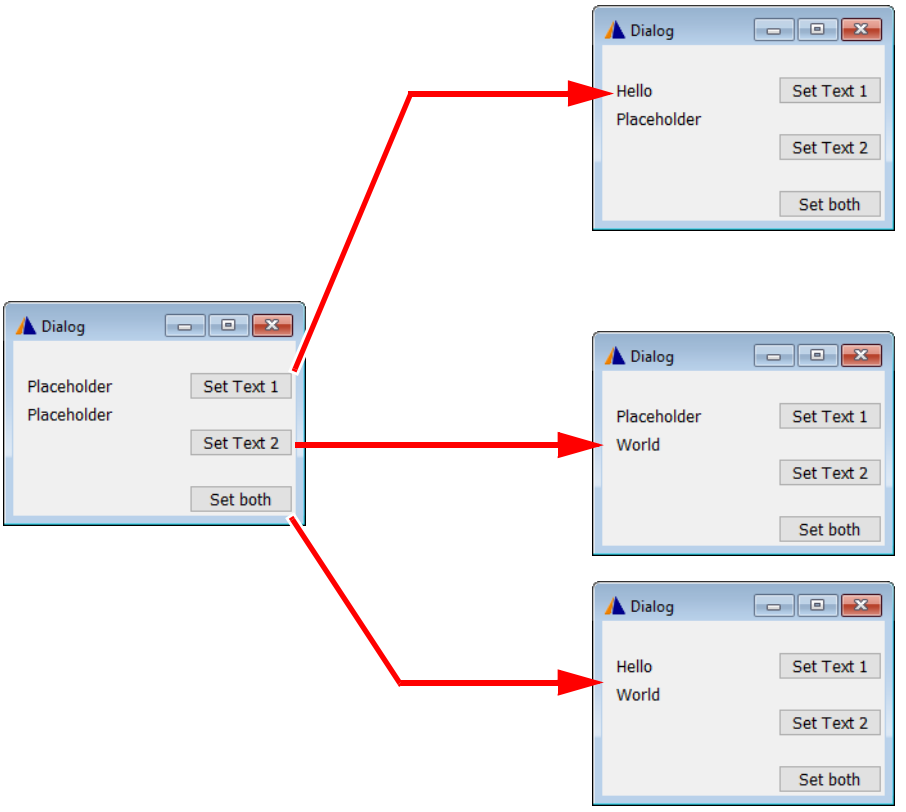
©1989-2020 Lauterbach GmbH

DIALOG.EXecute コマンドを使用し、コントロールに関連付けられたコマンドやブロックを実行することが可能です。

DIALOG.EXecute *<ラベル>*	*<ラベル>* に関連付けられたコントロールのコマンドを実行します。

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

STOP
DIALOG.END
ENDDO
```

©1989-2020 Lauterbach GmbH



©1989-2020 Lauterbach GmbH

ファイルのブラウジング

PRACTICE には、ファイルやディレクトリを参照するためのコマンドがいくつか用意されています。サブルーチンのように値を返すコマンドと、返された値をラベルに設定できるコマンドあります。.

DIALOG.DIR *<ディレクトリ名>*	ディレクトリ名を選択するためのダイアログボックスを作成します。*<ディレクトリ名>* にはワイルドカード文字を含める必要があります。
DIALOG.File *<ファイル名>*	ファイル選択ダイアログボックスを作成します。*<ファイル名>* にはワイルドカード文字を含める必要があります。また、*<ファイル名>* が存在し、*<ファイル名>* のリードのアクセス権をユーザが持っている必要があります。このコマンドで返されるファイル名は、ディレクトリを含んだフルパスになります。
DIALOG.File.SAVE *<ファイル名>*	ファイル選択ダイアログボックスを作成します。*<ファイル名>* にはワイルドカード文字を含める必要があります。*<ファイル名>* は存在しなくても構いませんが、選択したディレクトリにファイル作成のアクセス権をユーザが持っていなければなりません。このコマンドで返されるファイル名は、ディレクトリを含んだフルパスになります。

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

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

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

上記の例では、ファイル名とパス名を抽出するため、OS.FILE.NAME() 関数、OS.FILE.PATH() 関数を使用しています。

©1989-2020 Lauterbach GmbH

DIALOG.SetDIR *<ラベル><ディレクトリ名>*	ディレクトリ選択ダイアログボックスを作成します。*<ディレクトリ名>*にはワイルドカード文字を含める必要があります。このコマンドの実行結果は、*<ラベル>*に格納されます。
DIALOG.SetFile *<ラベル><ディレクトリ名>*	ファイル選択ダイアログボックスを作成します。*<ファイル名>*にはワイルドカード文字を含める必要があります。また、*<ファイル名>*は存在し、*<ファイル名>* のリードのアクセス権をユーザが持っている必要があります。このコマンドの実行結果は、*<ラベル>*に関連付けられた **EDIT** コントロールに設定されます。
DIALOG.SetFile.SAVE *<ラベル><ディレクトリ名>*	ファイル選択ダイアログボックスを作成します。*<ファイル名>*にはワイルドカード文字を含める必要があります。*<ファイル名>*は存在しなくても構いませんが、選択したディレクトリにファイル作成のアクセス権をユーザが持っていなければなりません。このコマンドの実行結果は、*<ラベル>*に関連付けられた **EDIT** コントロールに設定されます。

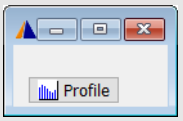
```
LOCAL &fname
DIALOG.view
(
      POS 1. 1. 25. 1.
FILE: EDIT "" ""
      POS 30. 1. 5. 1.
B:    BUTTON "..." "DIALOG.SetFile FILE *"
      POS 30. 3. 5. 1
      BUTTON "OK" "CONTinue"
)
STOP
&fname=DIALOG.STRing(FILE)
DIALOG.END
PRINT "File - &fname"
ENDDO
```

上記のスクリプト例では、**DIALOG.SetFile** コマンドが BUTTON コントロールにアタッチされています。コントロールの種類によっては、コマンドをアタッチすることが可能なものもあります。コマンドが 1 行の場合は、引用符 (") で囲んで記述します。複雑なコマンドの場合は、丸括弧 "()" で囲み PRACTICE ブロックで記述します。

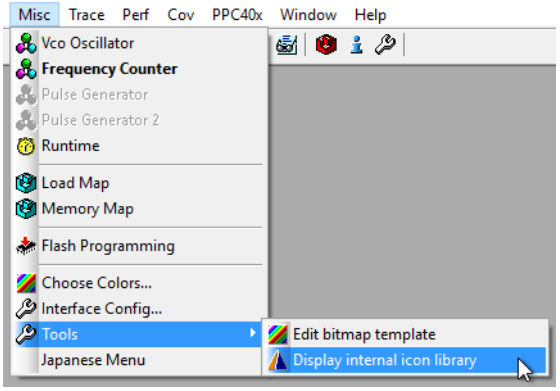
©1989-2020 Lauterbach GmbH

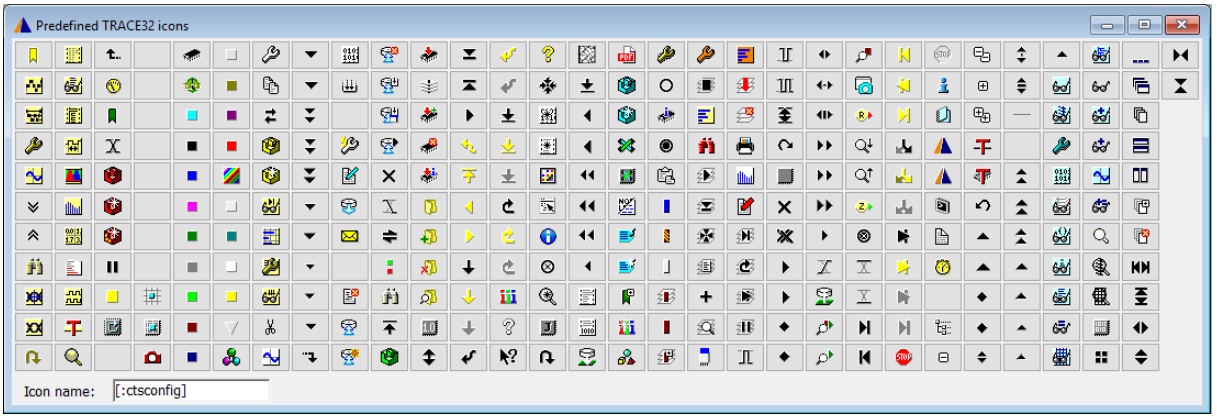
TRACE32 では、ダイアログやメニューのプログラミングで使用することができるいくつかの組み込みアイコンがあります。これらは、コントロールのテキスト部分に [:<**アイコン名**>] という構文を使って追加することができます。

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



内部のアイコンの一覧を TRACE32 上で表示することができます。アイコンを選択するとアイコン名が表示されます。





©1989-2020 Lauterbach GmbH

ダイアログの例

ダイアログのコントロールの完全なリストは、PowerView Command Reference (ide_ref.pdf) を参照してください。本節では、ダイアログのコントロールのうち、いくつかの使い方を紹介します。また、以下のスクリプトのフルバージョンは以下にあります。~~/demo/practice/dialogs/dialog_example_generic.cmm

まず、スクリプトで使用するローカルマクロを複数宣言し、デフォルト値を設定します。行によっては、読みやすさのために、"¥" 文字を使って行を分割しています。また、代入式は、**&instr** のように複数の代入式を使い、より長い値を生成しています。

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

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

&expand=0.
&progress=0.

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

以下のコードでは、ヘッダを作成し、ダイアログにテキスト行と編集フィールドを追加し、ダイアログを表示しています。開き括弧の後にある "&+" という文字は、ランタイムマクロをダイアログのコントロールに置き換えています。

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

次の記述は、ボックスとラジオボタンのグループ **CHOOSEBOX** をボックスペインに追加しています。各 **CHOOSEBOX** は同じグループの一部になります。ラベル名は、< グループ >.< メンバー > という形式をとり、各グループ中 1 つのメンバーのみに選択可能です。また、各 **CHOOSEBOX** は、コントロールが選択された際に、関連付けられたコマンドを実行します。

©1989-2020 Lauterbach GmbH

コマンドの実行中のマクロ置換を無効にするため、コマンドの先頭の括弧の後に「&-」という文字を記述しています。この記述により、コマンドで使用されているマクロの二重置換を防ぐことができます。各コマンドブロックは、そのグループの選択が既に行われていないかを確認し、選択されていない場合はリストを表示します。選択が行われている場合は、項目のリストが表示され、選択された項目がハイライト表示されます。これは、DIALOG.Set コマンドの２種類のフォームで行うことができます。

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

以下のコードは、リストを描画し、最初に &bond リストを生成しています。リスト内の項目がダブルクリックされると、コマンドが実行され、再び、マクロの置換は一時的に無効になります。CHOOSEBOX グループの現在のメンバーが決定され、そのグループからの選択のためのマクロが更新されます。その後、CHECKBOX の１つがチェックされ、グループ内からの選択が行われたことを示します。進捗率を含むマクロが更新され、これを使用してプログレスバーが更新されます。進捗率が９９%を超えると、ダイアログの OK ボタンが DIALOG.Enable を使って有効になります。

©1989-2020 Lauterbach GmbH

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

以下のコードでは、３つの CHECKBOX とプログレスバーを備えたボックスを描画しています。CHECKBOX コントロールは無効化され、スクリプトを介してのみ更新され、意図しないユーザ入力を防止しています。

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

©1989-2020 Lauterbach GmbH

以下のコードでは、ダイアログのサイズを拡大、縮小するために TREEBUTTON を使用しています。これにより、隠ぺいされた命令セットを可視化しています。

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

          )
```

このコードでは、TREEBUTTON 横に TEXTBUTTON を追加し、クリック領域を拡張しています。TEXTBUTTON は、DIALOG.EXecute を使って TREEBUTTON. のコマンドを実行します。

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

このコードでは、″OK″ というテキストを備えたボタンを描画し、クリックされると CONTinue コマンドを実行します。次に、マクロ &instr に格納されたテキストを表示する INFOTEXT コントロールを作成しています。INFOTEXT のオプションでは、ボーダースタイル、フォント、背景色が設定可能です。閉じ括弧は、ダイアログコントロールの終了を示しています。

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

©1989-2020 Lauterbach GmbH

ダイアログコントロールの終了後、いくつかのコントロールの初期状態を設定が必要なことがあります。CHECKBOX コントロールは無効化され、CHOOSEBOX のグループの初期メンバーが設定されています。ダイアログは、INFOTEXT コントロールを非表示にするためにサイズ変更され、最後の OK ボタンは無効化されています。

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

ユーザの名前が EDIT コントロールより取り出され、選択肢を示すメッセージボックスを表示しています。メッセージボックスが閉じられるとダイアログ (DIALOG.OK)、自体も閉じられ、スクリプトは終了します。

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

©1989-2020 Lauterbach GmbH

マルチコア環境での PRACTICE の利用

PRACTICE スクリプトをマルチコア構成のデバッグ環境において効果的に機能させるためには、若干の工夫が必要です。本セクションでは、マルチコア環境で効率的なデバッグを実現するためのノウハウと指針を示します。
ターゲットが SMP(Symmetric Multi-Processing) 用に設定されている場合、１つの PowerView インスタンスですべてのコアをコントロール可能です。SMP の場合、多くの PRACTICE コマンドでは、特定のコアを指定してコマンドを実行することを示すために、**/CORE <n>** オプションを準備しています。
本セクションでは、複数種類のコアで構成された AMP(Asymmetric Multi-Processing) システム上において、それぞれのコア上で PowerView インスタンスを起動し PRACTICE を利用する方法について説明します。

単純なマルチコア構成のデバッグでは、PowerView の１つのインスタンスがメインコアと接続後、別のプロセッサデバッグ用に２つめの PowerView インスタンスを起動、接続することができます。このような構成のシステムでは、以下の例のようにデバッグをすることが可能です。

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

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

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

ENDDO
```

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

マルチコアの構成によっては、特に PowerView インスタンス間の通信を要する場合など、より柔軟かつ複雑なアプローチが必要です。

©1989-2020 Lauterbach GmbH

インスタンス間の通信

PowerView では、各インスタンスが UDP を介して相互に通信できる INTERCOM と呼ばれる通信システムを備えています。INTERCOM についての詳細は、PowerView Command Reference (ide_ref.pdf) の "InterCom" を参照してください。INTERCOM を有効にするには、設定ファイル (通常は ~~/config.t32) を以下のように変更します。(主な変更点を赤字で示しています。)

```
PBI=
USB
CORE=1

IC=NETASSIST
PORT=20001
```

PORT= の記述は、PowerView インスタンスが、この UDP ポート番号上の着信リクエストをリッスンすることを指示します。ポート番号はユニークである必要があります。

複数の PowerView インスタンスを利用する場合は、それぞれのインスタンス用に、固有の CORE=、PORT= を指定する個別の設定ファイルが必要になります。また、共通の設定ファイルを利用する場合は、それぞれのインスタンス起動時にコマンドライン引数として値を指定することも可能です。(以下に例を示します。)

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

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

以下のコマンドラインで PowerView を起動します。

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

©1989-2020 Lauterbach GmbH

以下は、PRACTICE スクリプトから PopwerView インスタンスを起動する例です。

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

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

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

ENDDO
```

上記の例では、メインのスクリプトは、InterCom.WAIT コマンドを使用して、他の PowerView インスタンスが利用可能になるまで待っています。

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

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

ENDDO
```

すべての PowerView インスタンスが同期した後、各インスタンスにコマンドを発行することが可能です。コマンドの発行は、InterCom.execute コマンドを使用して実行可能です。以下の例では、PRACTICE マクロでラップしスクリプトの可読性をあげています。

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

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

©1989-2020 Lauterbach GmbH

ロバスト性を意識した PRACTICE スクリプトの設計

スクリプト開発者の環境で実行可能なスクリプトでも、他のメンバや顧客の環境上では正常に動作しないことがあります。このセクションでは、より移植性が高く、かつ堅牢な PRACTICE スクリプトの設計方法を示します。

移植性が高く、堅牢な PRACTICE スクリプトを設計する際は、下記のような点に留意してください。

- ホスト PC のディレクトリ構造が異なる可能性
- ホスト PC の OS、またはそのバージョンが異なる可能性
- 異なるデバッガハードウェアを使用している可能性
- スクリプトが、異なるターゲットボード、プロセッサ上で実行される可能性
- TRACE32 デバッガのバージョンの相違
- TRACE32 デバッガの設定の相違

©1989-2020 Lauterbach GmbH

ディレクトリパス名の扱い

スクリプト内ではパス名をハードコーディングせず、ネットワークドライブは、別の文字にマッピングすることなどを意識します。

以下は、ELF ファイルが現在の作業ディレクトリにある場合にのみ動作する例です。

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

以下の例は、ELF ファイルが、特定のパスにある場合にのみ動作し、プロジェクトを C:¥users¥barry¥Project_1¥ に保存している場合は正しく動作しません。

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

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

OS.PresentWorkingDirectory()	作業ディレクトリの名前を文字列で返します。

移植性を高める方法としては、相対ディレクトリパスを使用する方法があります。TRACE32 では、以下に示すように多種の関数や組み込みのディレクトリ取得方法を用意しています。

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

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

OS.PresentPracticeDirectory()	現在の PRACTICE スクリプトが置かれているディレクトリの名前を文字列で返します。 ~~~~/ は、TRACE32 のパスプレフィックスに相当します。

©1989-2020 Lauterbach GmbH

スクリプトが ELF ファイルと同じディレクトリにある場合、前記の例は正常に動作します。また、以下の例では、ELF ファイルをサブディレクトリに配置した場合の記述です。

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

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

読み込まれる ELF ファイルの指定にワイルドカードを利用することができます。

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

以下のスクリプトでは、プロジェクトのトップレベルのディレクトリパスの一部として利用しています。

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

©1989-2020 Lauterbach GmbH

ロバスト性の高いスクリプトにするためには、ファイルの存在を確認した後、存在しない場合には、ユーザにその旨を通知するなどの処理を加えます。

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

OS.FILE(*<ファイル名>*)　　*<ファイル名>*が存在する場合に TRUE を返します。

ファイルパス取得関連機能の完全なリストは、PowerView User's Guide (ide_user.pdf) の **"Path Prefixes"** を参照してください。

©1989-2020 Lauterbach GmbH

ホスト OS

ホスト OS は様々な差異があるため、移植性の高いスクリプトは、これらの差異がユーザに与える影響を考慮して記述する必要があります。

スクリプトがパス名区切り文字としてスラッシュ (″/″) を使用している場合、TRACE32 は自動的にホスト OS の正しい区切り文字に変換します。

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

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

ファイルシステムの操作には、TRACE32 の組み込みコマンドが使用されます。組み込みコマンドは、ホスト OS のコマンドに自動的に変換され、実行されます。組み込みコマンドの完全なリストは、PowerView User's Guide (ide_user.pdf) の **"File and Folder Operations"** を参照してください。

REN *<ファイル名>*	ファイルをリネームします。
COPY *<ソース> <デスティネーション>*	ファイルをコピーします。
MKDIR *<パス名>*	ディレクトリを作成します。
RMDIR *<パス名>*	ディレクトリを削除します。
ZIP *<ソース>[<デスティネーション>]*	ファイルを圧縮します。

TRACE32 は、″~~~″ をシステムディレクトリに展開します。この変換を利用し、ホスト OS の差異を吸収した記述を行うことができます。

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

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

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

行末を示す終端コードは、OS によって異なるコードが使用されています。

- Windows では CR + LF を使用します。
- Linux 、MacOS では、LF を使用します。

©1989-2020 Lauterbach GmbH

TRACE32 では、LF が行末を示すコードとして認識します。以下のコードでは、任意のホスト上のファイルから 1 行ずつ読み込みます。

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

TRACE32 からファイルへの書き込む際は、現在のホスト OS の正しい終端コードを使用します。このため、スクリプトからの書き込み処理では、ホスト OS の種類を意識する必要がありません。

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

関数 OS.VERSION(0) を使用して、ホストオペレーティングシステムを決定することができます。この機能を利用して、スクリプトは OS の違いに対応することができます。

デバッガハードウェア

スクリプトが、特定の TRACE32 ハードウェアで動作しない、異常動作するなど、利用する TRACE32 ハードウェアに依存する場合があります。このため、スクリプトは 利用が想定される TRACE32 ハードウェア上での動作確認後に配布する必要があります。スクリプトには、TRACE32 ハードウェア検出のための関数が多数存在します。詳細は、General Function Reference (general_func.pdf) を参照してください。以降では、そのうちのいくつかの記述例を示します。ハードウェア検出関数は、関連するハードウェアが検出された場合に TRUE を返します。

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

©1989-2020 Lauterbach GmbH

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

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

ターゲット CPU/ ボード

関数 CPUFAMILY() は、動作している CPU のファミリの名前を返します。同じ t32m* バイナリファイルで、デバッガが起動可能なコアはすべて同一のファミリです。各ファミリには、それぞれ独自の関数セットがあります。

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

関数 CPU() は、SYStem.state ウィンドウ内に選択されているプロセッサの値を返します。

©1989-2020 Lauterbach GmbH

関数 IDCODE(<n>) を使用して、JTAG スキャンチェーン上の n 番目のデバイスの JTAG ID コードを取得することができます。

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

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

CPU によっては、１６、２４、３１ビットなどのアドレッシングのみを受け付け、８ビット（バイト）でのアドレッシング指定ができません。以下のスクリプトは、アドレッシングのビット値を調べる例を示しています。

```
LOCAL &width

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

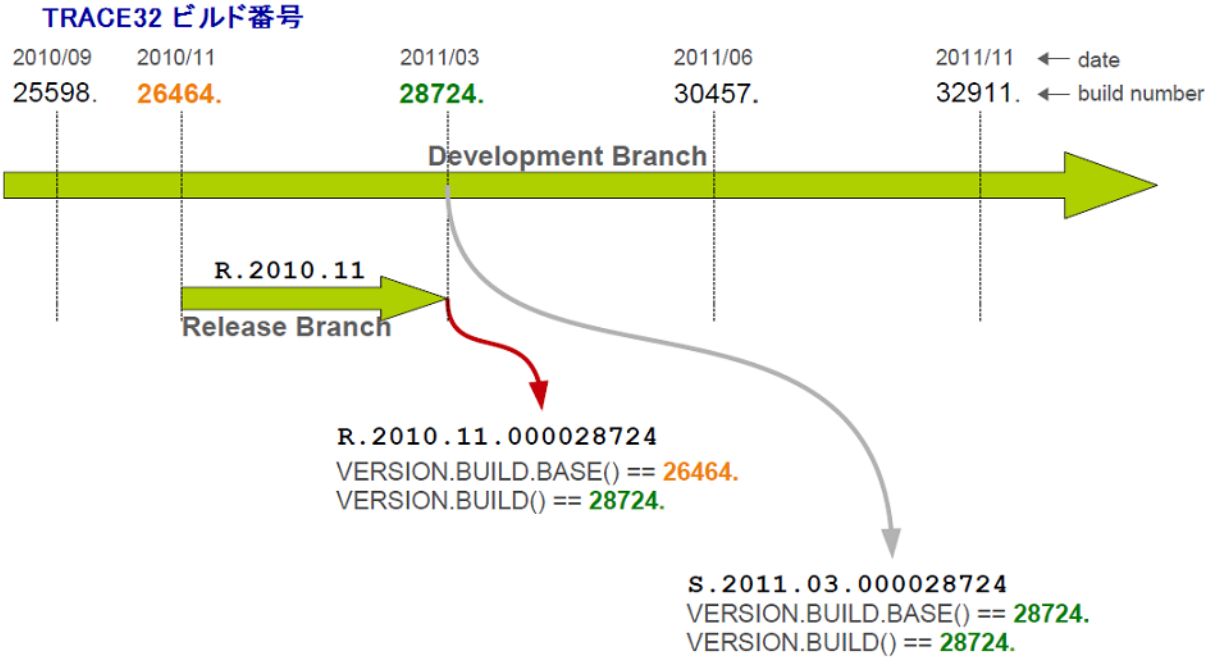
©1989-2020 Lauterbach GmbH

TRACE32 PowerView バージョン

スクリプトからは、TRACE32 PowerView のバージョン依存機能の記述は避けるべきです。以下の関数は、TACE32 PowerView のビルド番号を確認する関数です。

VERSION.BUILD()	TRACE32 PowerView のビルド番号を返します。このビルド番号までにリリースされている機能がすべて網羅されているわけではありません。
VERSION.BUILD.BASE()	TRACE32 PowerView のビルドのベースとなるビルド番号を返します。このビルド番号までにリリースされたすべての機能が含まれます。

以下の図は、上記の２つの異なるビルド番号の関係を説明しています。




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

TRACE32 設定

RACE32 はユーザによるカスタマイズ性が高いため、スクリプトはユーザの設定に依存しないように記述する必要があいます。

©1989-2020 Lauterbach GmbH

以下は、数値の基数の指定は、環境ごとに設定可能であるため、基数の差異により実行結果が異なる例です。

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

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

スクリプトは、このような混乱を避けるために、基数を必ず指定すべきです。

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

PRACTICE マクロを出力する前にスクリプトでフォーマットして、結果が環境依存にならないようにすることを推奨します。以下にいくつかのスクリプトの記述例を示します。.

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

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

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

出力フォーマットを行う関数は、FORMAT.HEX()、FORMAT.Decimal() 、FORMAT.BIN() の３種類が用意されています。

©1989-2020 Lauterbach GmbH

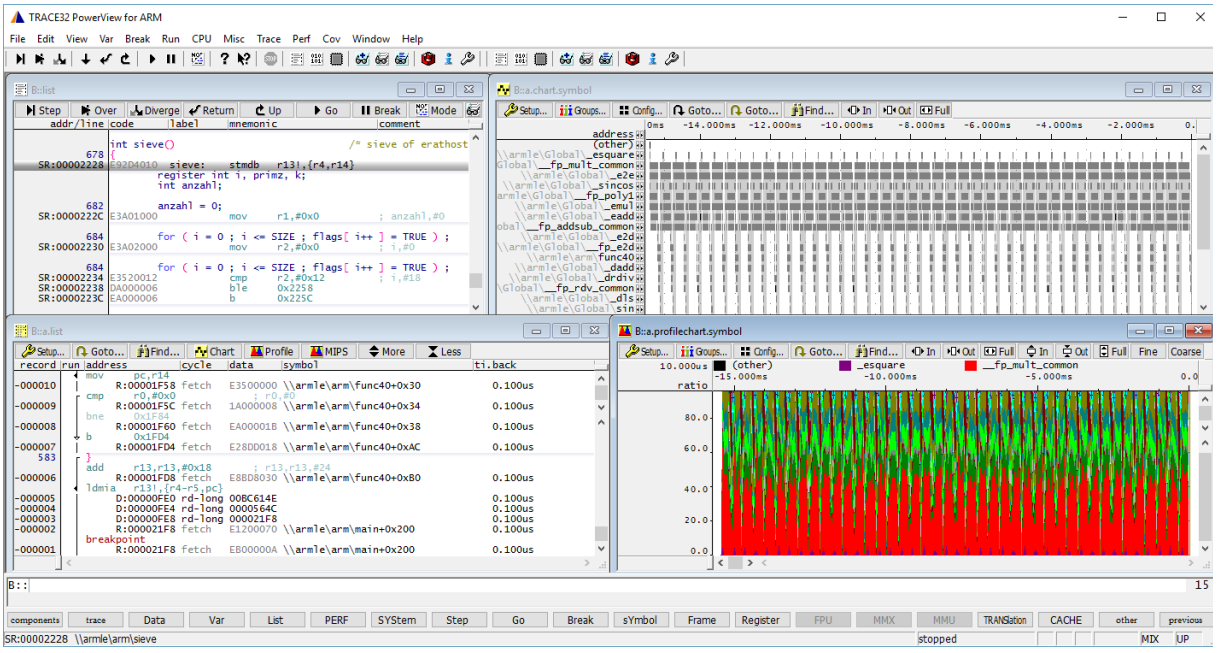
複数のブレークポイントを利用したり、複数のウィンドウ利用するような一連の手順をスクリプト化する場合は、作業が完了した後にデバッガの設定を保存する STOre コマンドを使用します。

```
PRIVATE &bpfile &winfile

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

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

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

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

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

; Now clear all the existing Windows
WinCLEAR


; Run the rest of the script here


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

©1989-2020 Lauterbach GmbH

スクリプトの実行結果表示用に専用のウィンドウを開く必要がある場合、解像度、サイズを特定して動作させることは推奨されません。ウィンドウ位置、サイズを絶対値で指定した場合、環境によっては判読しやすい表示にならないことがあります。**WinPOS** コマンドは、絶対値だけでなくパーセンテージも引数に取ることができます。現在の TRACE32 ウィンドウのサイズ、位置を基準としてパーセンテージで、サイズ、位置を指定することができます。

```
WinCLEAR

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

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

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

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

ENDDO
```



別の手段として、スクリプト中で画面解像度を取得し、その画面サイズに適合するウィンドウを開く方法（以下）があります。また、この方法のスクリプト例は、~~/demo/practice/screen_size.cmm でも提供しています。

©1989-2020 Lauterbach GmbH

```
PRIVATE &scrw &scrh

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

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

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

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

ENDDO
```

©1989-2020 Lauterbach GmbH

PRACTICE スクリプトがエラーに遭遇した場合、エラーが発生した行で実行を停止します。SETUP.WARNSTOP コマンドを使用すると、警告が出たコマンドで実行を停止するように設定することもできます。PRACTICE スクリプトのより高度なエラー処理を使用すると、エラーが発生した際に、エラーから復帰するように記述することができます。これは PRACTICE スクリプトのイベント駆動機能に依存しています。以下は、最も単純な利用例で、エラー発生後もスクリプトの実行を継続しています。

```
; Enable error handler
ON ERROR GOSUB RETURN

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

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

ON ERROR inherit の呼び出しは、直前のエラーハンドラを復元しています。エラーハンドラはスタックされている可能性があるため、本来のエラーハンドラは次のよう記述します。

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

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

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

©1989-2020 Lauterbach GmbH

ビルド 72159 以降の TRACE32 PowerView では、各コマンドの実行後にエラーが発生の有無を確認できるようになりました。この機能により、PRACTICE スクリプトは、実践的なエラーハンドラーを記述して、スクリプトが移乗動作しないようにする記述が可能です。

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

ON ERROR GOSUB
RETURN

; label to return to for re-try
start_here:

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

ENDDO
```

©1989-2020 Lauterbach GmbH

再利用可能なスクリプトを設計するには、渡される引数にも留意する必要があります。スクリプトの引数が常に正しい値であるとは限らないため、実行時に引数の値を確認することが重要です。
STRing.SCANAndExtract() を利用して、引数の順序の正当性を確認できますが、引数の内容を保証することはできません。PRACTICE マクロの場合は、常に文字列として扱われ、以下の例の IsNum サブルーチンのように、マクロが１０進数または、１６進数形式の数値だけを含むか否かを確認することができます。

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

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

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

ENDDO

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

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

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

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

©1989-2020 Lauterbach GmbH

独自コマンドの作成

PowerView では、独自のカスタムコマンドを作成することができます。独自コマンドは、GLOBALON コマンドを用いて定義します。

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

コマンドには、*<name>* を命名できますが、文字長は最大９文字までに制限されます。以下の例では、引数なしで呼び出された場合は、コマンド **TESTRUN** を登録し、引数１つの **"REMOVE"** で呼び出してコマンド **TESTRUN** を削除します。

```
PARAMETERS &test &testfile &count

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

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

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

©1989-2020 Lauterbach GmbH

コマンドが登録されると、TRACE32 コマンドラインや、他のスクリプト中から登録したコマンドが実行できます。

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

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

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

間違いやすい記述

この最後のセクションでは、PRACTICE スクリプトの記述方法でよくある間違いをいくつか取り上げています。

スクリプトをデバッグする場合を除き、ブロッキングコマンドは使用しないようにしてください。STOP や ENTER のようなコマンドはスクリプトの実行を停止しますが、ユーザにはスクリプトが停止していること、入力を待ちになっていることを知らせる手段がありません。このような場合、ダイアログの表示を使用するのが推奨されます。

スペースの挿入には注意してください。これは IF または WHILE コマンドの後に必ず必要ですが、式の中にはスペースを記述しないように留意してください。.

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

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

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

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

©1989-2020 Lauterbach GmbH

関数の引数にファイル名を含まないようにしてください。

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

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

常に PRACTICE マクロは正しく宣言してください。特に、LOCAL と PRIVATE の違いに留意してください。

デフォルトの基数が期待している状態でないと仮定して、かならず基数を指定してスクリプトを記述するようにしてください。例えば、ただ単に ″10″ と記述せず、″0x10″ または、１０進表記であれば、″16.″ と記述します。

スクリプトやファイルへのパスをハードコーディングする代わりに、ディレクトリパス取得用の組み込みコマンドを使用して記述します。

ホスト OS のファイル操作（コピー、削除、ディレクトリ移動など）には、ファイル操作用の組み込みコマンドを利用してください。異なるホスト OS 間でもスクリプトの移植容易性を保ちます。

©1989-2020 Lauterbach GmbH