


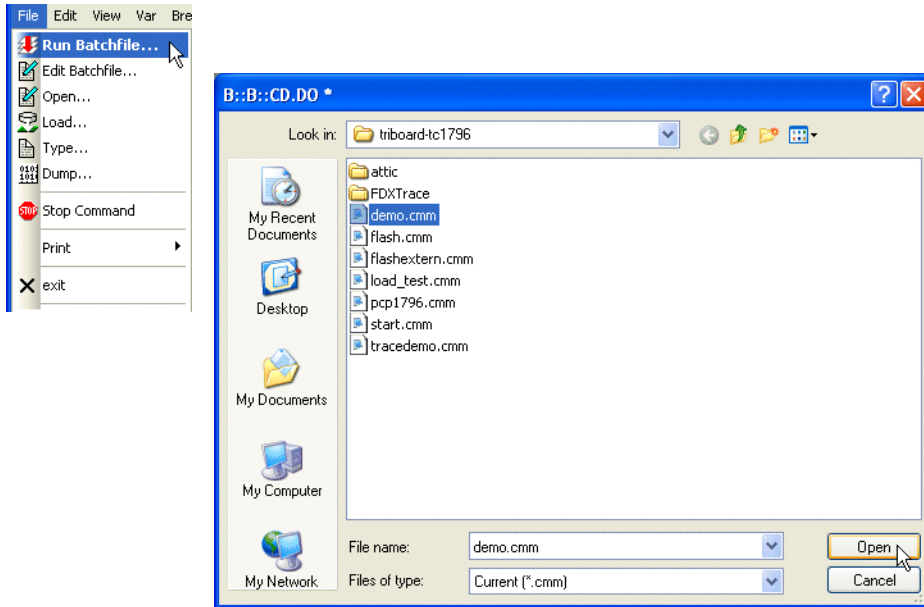
TRACE32 Online Help	
TRACE32 Directory	
TRACE32 Index	
TRACE32 Training	
PRACTICE - トレーニング	2
PRACTICE - バッチ言語の概要	2
バッチファイルの実行	3
PRACTICE プログラムの作成	4
設定の保存	4
LOG コマンド	8
コマンド履歴	9
エディタ PEDIT	10
PRACTICE プログラムのデバッグ	11
PRACTICE エディタ PEDIT からデバッガを起動	11
PRACTICE スタックの表示	12
デバッグコマンド	13
テスト中の画面更新	14
PRACTICE プログラムの構造	15
プログラム要素	15
コマンド	15
関数	16
コメント	18
ラベル	18
プログラムフロー制御	19
別の PRACTICE プログラム内での PRACTICE プログラムの起動	19
その他のコマンド	21
PRACTICE プログラムまたはサブルーチンにパラメータを渡す	22
条件付きプログラム実行	24
PRACTICE 変数	26
変数の内容	27
PRACTICE コマンド実行と変数	31
型変換	35
文字列の演算	35
I/O コマンド	36
DIALOG プログラミング	40
ファイルコマンド	42
PRACTICE を介したイベント制御	44
索引 (ローカル)	45

PRACTICE - バッチ言語の概要

以下にバッチファイルの主なタスクを示します：

- 開発ツールに適切な起動シーケンスを提供する
- FLASH プログラミングを自動化する
- ユーザーインターフェースをカスタマイズする
- 特定の設定を保存して再現する
- 自動テストを実行する

バッチファイルの標準的な拡張子は `.cmm` です。



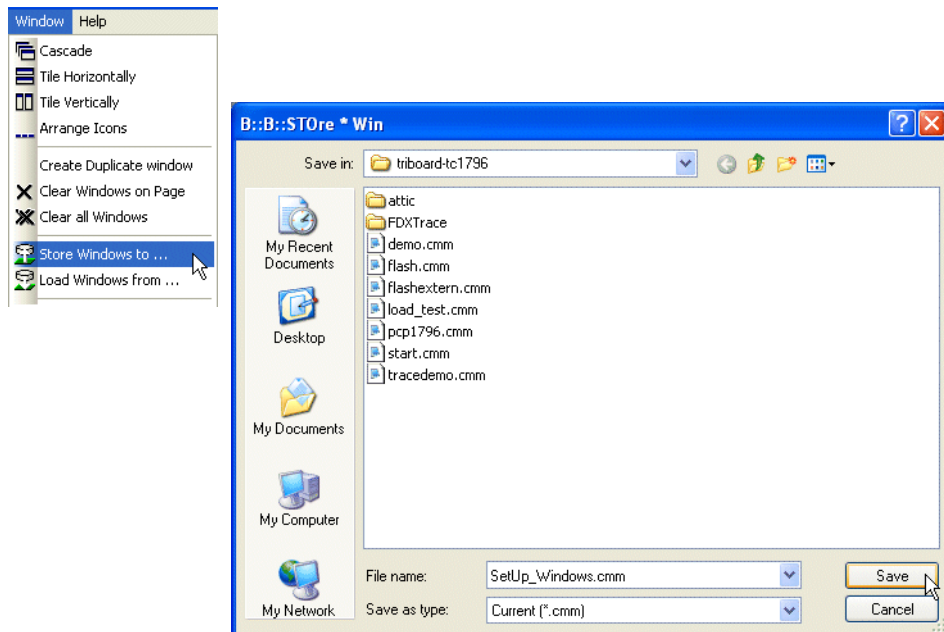
- DO** < ファイル名 > [< パラメータリスト >] バッチを開始します。
- ChDir.DO** < ファイル名 > [< パラメータリスト >] バッチのあるディレクトリに移動してバッチを開始します。
- RUN** < ファイル名 > [< パラメータリスト >] バッチのテスト用：PRACTICE スタックをクリアしてからバッチを開始します。
- PATH** [+>] < パス名 > PRACTICE スクリプトの検索パスを定義します。

```
DO memtest 1000--2000  
ChDir.DO c:¥t32¥demo¥powerpc¥diabc  
ChDir.DO g:¥t32¥test¥memtest 1000--2000  
PATH c:¥t32¥test
```

設定の保存

STOre コマンドでバッチファイルを作成して、選択した設定をいつでも再現できます。

現在のウィンドウ設定の保存



STOre < ファイル名 > Win | WinPAGE

バッチを作成して現在のウィンドウ設定を復元します。

ClipSTOre Win | WinPAGE

コマンドを作成し、現在のウィンドウ設定を復元して cliptext で提供します。

```
; Result of the command STOre win1 Win

// And Mon Aug 06 10:37:08 2001

B::

TOOLBAR ON
STATUSBAR ON
WINPAGE.RESET

WINCLEAR
WINPOS 0.0 13.25 34. 18. 5. 0. W001
Var.Frame /Locals /Caller

WINPOS 37.625 19.75 81. 12. 10. 1. W002
WINTABS 32.
TRace.List

WINPOS 37.375 13.1875 81. 4. 0. 0. W003
Var.AddWatch ¥¥diabp8¥Global¥ast

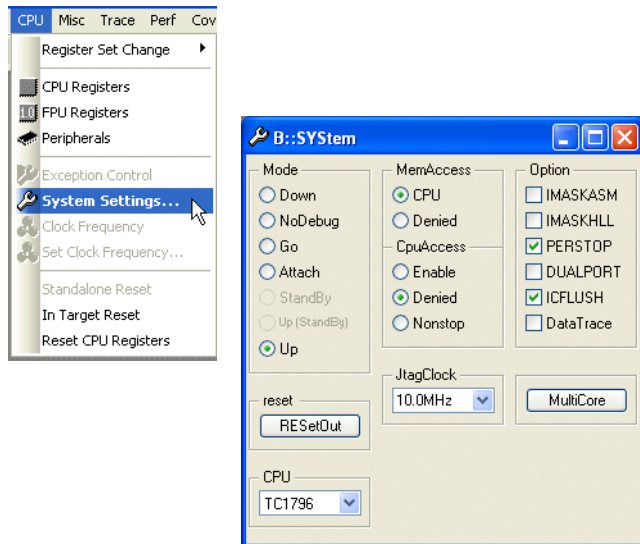
VAR.ADDWATCH ¥¥diabp8¥Global¥flags

WINPOS 82.0 0.0 69. 9. 23. 1. W004
WINTABS 0. 0. 0.
Break.List

WINPOS 0.0 0.0 79. 9. 13. 1. W000
WINTABS 10. 10. 25. 62.
Data.List

WINPAGE.SELECT P000

ENDDO
```



STOre < ファイル名 >
SYStem

バッチを作成して SYStem 設定を復元します。

ClipSTOre SYStem

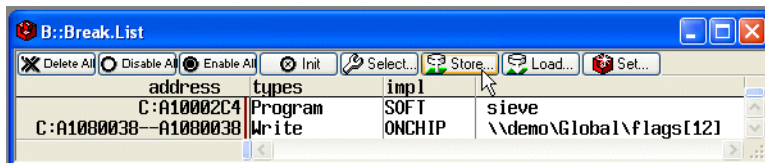
コマンドを作成し、現在の SYStem を復元して cliptext で提供します。

```
; Result of the command ClipSTOre SYStem
// andT32_1000039 Mon Jan 23 09:32:37 2006
```

```
B:::
```

```
SYSTEM.RESET
SYSTEM.CPU TC1796
SYSTEM.MEMACCESS CPU
SYSTEM.CPUACCESS DENIED
SYSTEM.JTAGCLOCK 10000000.
SYSTEM.OPTION PERSTOP ON
SYSTEM.OPTION DUALPORT OFF
SYSTEM.OPTION HEARTBEAT OFF
SYSTEM.OPTION ICFLUSH ON
SYSTEM.MODE UP
```

```
ENDDO
```



STore < ファイル名 > Break バッチを作成してブレークポイント設定を復元します。

ClipSTore Break コマンドを作成し、ブレークポイント設定を復元して cliptext で提供します。

```
// andT32_1000039 Mon Jan 23 09:37:59 2006

B::

BREAK.RESET
B.S sieve /P
V.B.S ¥¥demo¥Global¥flags[12]; /W

ENDDO
```

ブレークポイントはシンボリックレベルで保存されます。

LOG コマンド

LOG コマンドを使用して、TRACE32 ユーザーインタフェースのほとんどのアクティビティを記録できます。

LOG コマンドを制御するコマンド：

LOG.OPEN < ファイル >	LOG コマンド用のファイルを作成して開きます。LOG ファイルのデフォルトの拡張子は (.log) です。
LOG.CLOSE	LOG コマンド用のファイルを閉じます。
LOG.OFF	LOG コマンドを一時的に無効にします。
LOG.ON	LOG コマンドを再度有効にします。

```
LOG.OPEN log1          Creates and opens the command log file
...                    Record the command log
LOG.CLOSE              Close log
```

LOG コマンドの内容

```
// AndT32 Mon Jul 29 09:29:24 2002

// B::B::Var.View *
B::B::B::Var.View flags
B::B::Break.Set
B::B::
B::B::
B::B::Var.Break.Set (flags[3]); /W
B::B::Break.List
```



コマンド履歴にはコマンドラインに入力したコマンドだけが記録されます。HISTORY ファイルのデフォルトの拡張子は (.log) です。

HISTORY.type

コマンド履歴を表示します。

```
B::HISTORY
B::MAP.BOnchip 0xa0000000--0xa01ffffff
B::MAP.BOnchip 0xa0000000--0xa01ffffff
B::map.reset
B::MAP.BOnchip 0xa0000000--0xa01ffffff
B::Clipstore system
B::Break.Set sieve
B::v.b.s flags[121] /w
B::clipstore break
B::HISTORY
```

HISTORY.SAVE [< ファイル名 >]

コマンド履歴を保存します。

HISTORY.SIZE [< サイズ >]

コマンド履歴のサイズを定義します。

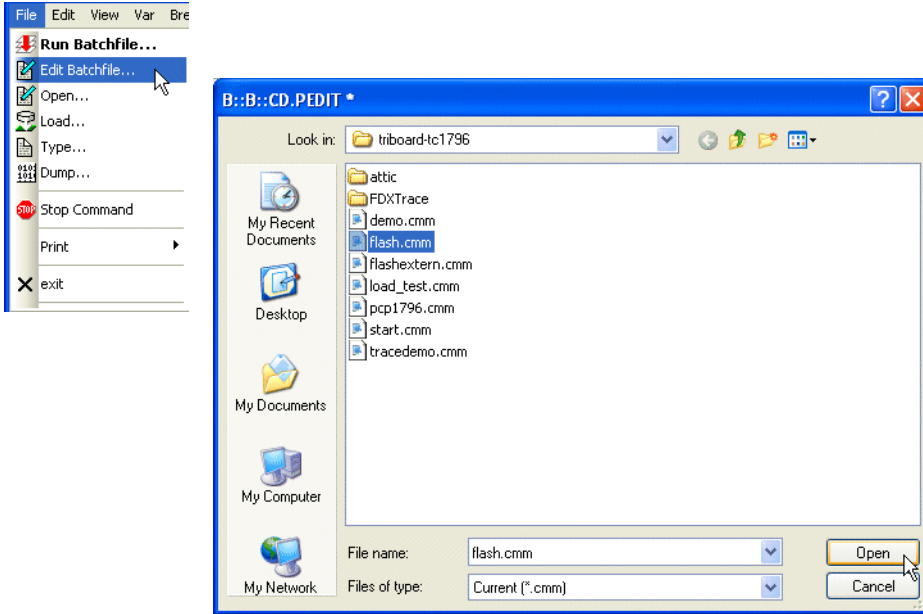
T32.cmm ファイルに以下の命令が記述されている場合：

```
AutoSTOre , HISTORY
```

TRACE32 を終了したときに、コマンド履歴が自動的に TMP ディレクトリに保存され、TRACE32 を起動したときに再コールされます。

AutoSTOre < ファイル名 > [< 項目 > ...]

TRACE32 の終了時に、定義した設定を自動的に保存します。



PEDIT のデフォルトの拡張子は (.cmm) です。

PEDIT.open < ファイル名 > [< ライン番号 >] [< オプション >] PRACTICE エディタを開きます。

PRACTICE プログラムを保存

新しい名前を付けて PRACTICE プログラムを保存

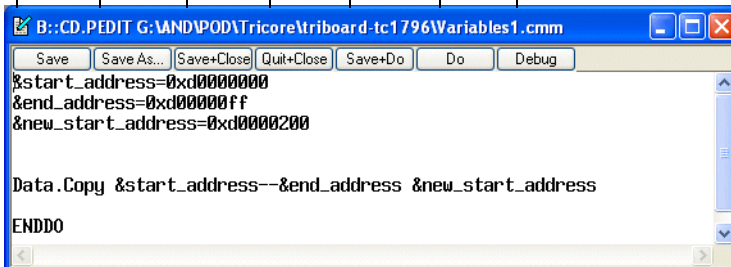
PRACTICE プログラムを閉じて保存する

PRACTICE プログラムを閉じるが保存しない

PRACTICE プログラムを保存して起動する

PRACTICE プログラムを起動

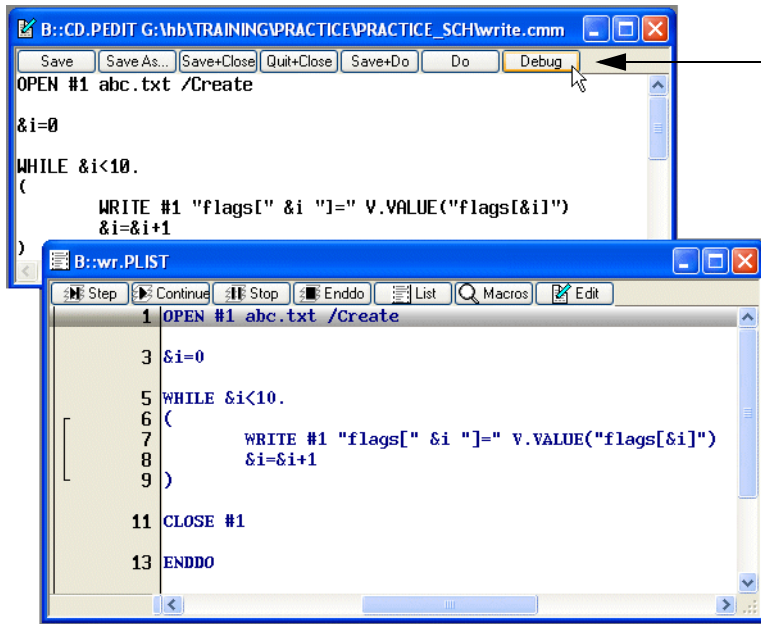
PRACTICE プログラムをデバッグ



PRACTICE プログラムのデバッグ

TRACE32 は PRACTICE プログラムのデバッグをサポートします。

PRACTICE エディタ PEDIT からデバッガを起動



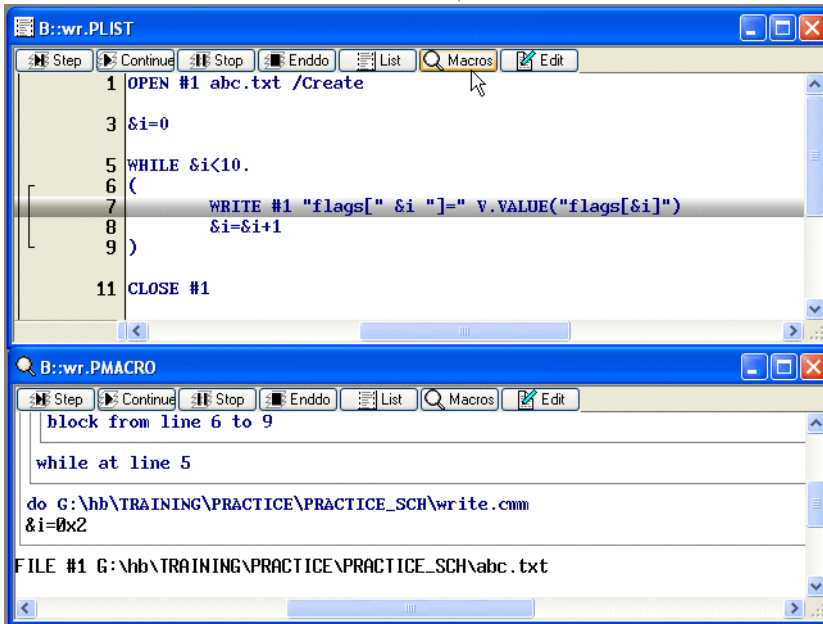
[Debug] を使用して PRACTICE プログラムのデバッガを起動します。

[PLIST] ウィンドウのローカルボタン

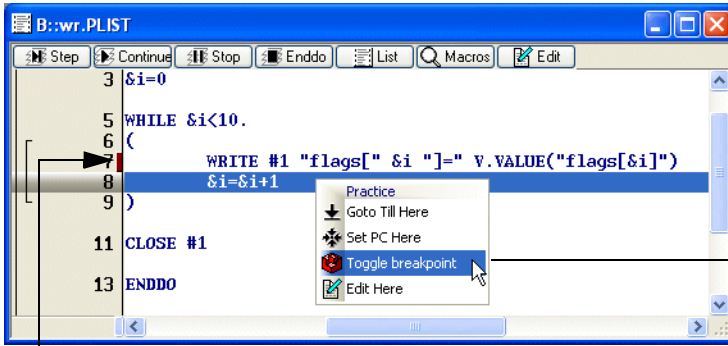
[Step]	PRACTICE プログラムを 1 ステップ実行します。
[Continue]	PRACTICE プログラムを実行します。
[Stop]	PRACTICE プログラムの実行を停止します。
[Enddo]	現在の PRACTICE プログラムを終了します。コール元の PRACTICE プログラムで実行が継続されます。コール元のプログラムが存在しない場合は、PRACTICE プログラムの実行が停止します。
[Macro]	PRACTICE スタックを表示します。
[Edit]	PEDIT を開いて PRACTICE プログラムを編集します。

PLIST [< ライン>] デバッグウィンドウに PRACTICE プログラムをリスト表示します。

PRACTICE スタックを表示します。



PRACTICE スタックにプログラムのネスト状態、ローカルおよびグローバル PRACTICE 変数が表示されます。



このラインは PRACTICE
ブレークポイント

PRACTICE デバッグプルダウン

[Goto Till Here]	選択したラインまで PRACTICE プログラムを実行します。
[Set PC Here]	PRACTICE PC を選択したラインに設定します。
[Toggle breakpoint]	PRACTICE デバッガでは 1 つのブレークポイントを使用できません。[Toggle breakpoint] でこのブレークポイントを設定または削除できます。
[Edit Here]	PEDIT を開いて PRACTICE プログラムを編集します。カーソルが選択したラインに自動的に移動します。



問題を修正した後で PRACTICE プログラムを再起動する場合は、**ENDDO** を使用して、呼び出した PRACTICE スクリプトをスタックから削除するか、**END** を使用して PRACTICE スタック全体をクリアします。PRACTICE スタックからグローバル変数を削除するには、**PMACRO.RESet** を使用します。

PRACTICE プログラムの実行中は、通常、画面は更新されません。

画面の更新を制御するには SCREEN コマンドを使用します。

SCREEN.ALways

PRACTICE 命令を実行するたびに画面を更新します。これにより実行速度が低下します。

SCREEN.ON

PRINT コマンドを実行するたびに画面を更新します。

SCREEN.display

すぐに画面を更新します。

SCREEN.WAIT

画面を更新してウィンドウ処理を待ってから、PRACTICE プログラム実行を続行します。

```
; PRACTICE file screen_wait.cmm          ; accumulate 10 program runs to
                                           ; one measurement result

Trace.Mode Leash                          ; stop program execution when trace
                                           ; buffer is full

&i=1.

WHILE &i<=10.
(
  PRINT "RUN " %Decimal &i
  Go                                       ; start program execution

  Wait !RUN()                             ; wait until trace buffer is full
                                           ; and program execution is stopped

  Trace.STATistic.Func /Accumulate       ; perform function run-time
                                           ; analysis

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

  &i=&i+1.
)
ENDDO
```

プログラム要素

コマンド

TRACE32 開発ツールのすべてのコマンド

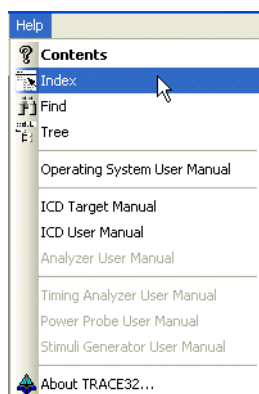
プログラムフロー制御用コマンドおよび条件付きコマンド

I/O コマンド

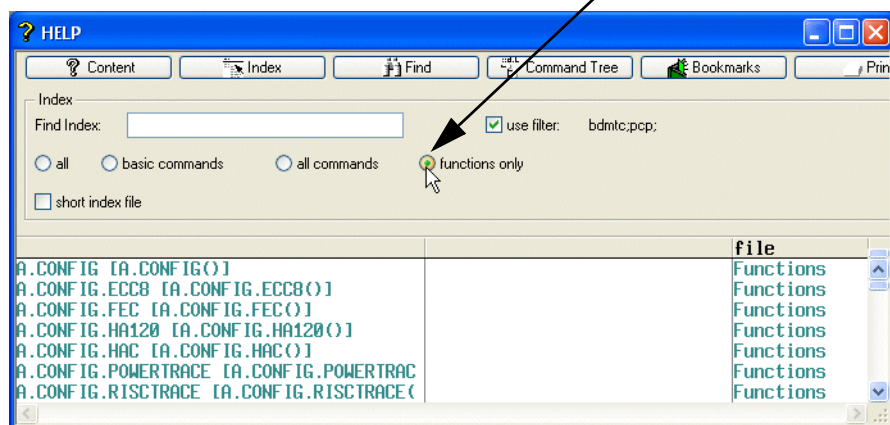
関数を使用して、ターゲットシステムの状態や開発ツールの状態についての情報を取得します。

Register(<名前>)	指定した CPU レジスタの内容を取得します。
V.VALUE(<HLL 式>)	HLL 式の内容を取得します。
RUN()	プログラムがターゲットで実行されている場合は true を返し、プログラム実行が停止した場合は false を返します。
OS.PWD()	現在の作業ディレクトリの名前を文字列で返します。
CONV.CHAR(<値>)	整数値を ASCII 文字に変換します。

使用可能な関数のリストはオンラインヘルプで参照できます：



[functions only] を選択



CPU()

選択したプロセッサの名前を文字列で返します。

```
PRINT CPU()

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

SYSTEM.UP()

デバッガと CPU が通信している場合は TRUE を返します。

RUN()

CPU がアプリケーションプログラムを実行している場合は TRUE を返します。

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

IF RUN()
    BREAK
```

VERSION.BUILD()

ビルド番号を返します。

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

Data.Byte (< アドレス >) < アドレス > の byte 値を返します。

Data.Word (< アドレス >) < アドレス > の word 値を返します。

Data.Long (< アドレス >) < アドレス > の long 値を返します。

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

STRING.UPR(< 文字列>) 大文字に変換された < 文字列 > を返します。

```
; script function_string.cmm  
  
DIALOG.FILE *.sre  
ENTRY &filename  
&filename=STRING.UPR("&filename")  
  
PRINT "&filename"  
  
ENDDO
```

コメント

コメントの先頭には「;」または「//」を付けます。

ラベル

最初の列にラベルを入力する必要があります。ラベルの最後には「:」を付けます。



不要な空白を使用しないことをお勧めします。不要な空白があると、PRACTICE コマンドが誤って解釈される場合があります。

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

別の PRACTICE プログラム内での PRACTICE プログラムの起動

PRACTICE プログラムが別の PRACTICE プログラムをコールすることができます。これにより、複数の PRACTICE プログラムを構成できます。

DO < ファイル名 > [< パラメータリ
スト >] 別の PRACTICE プログラム内で PRACTICE プログラムを起
動します。

```
; PRACTICE file modular.cmm

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

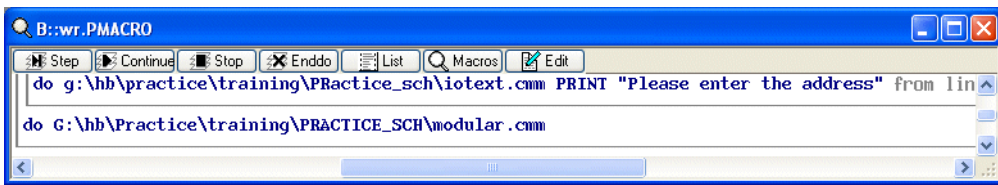
DO iotext                     ; Start PRACTICE script iotext

AREA.RESet                   ; Close IO window
WinCLEAR IO1

ENDDO
```

移動可能な PRACTICE スクリプトの書き込むときには、以下の省略形を使用できます：

Windows	機能
.¥	現在のディレクトリ
..¥	親ディレクトリ
~¥	ユーザーのホームディレクトリ（\$HOME から）
~~¥	TRACE32 のシステムディレクトリ
~~~¥	TRACE32 の一時ディレクトリ
~~~~¥	現在の PRACTICE スクリプトが置かれているディレクトリ



PRACTICE は、PRACTICE スタックの PRACTICE プログラムのコール階層を管理します。ローカルおよびグローバル PRACTICE 変数もこのスタックに置かれます。



各 PRACTICE プログラムを **ENDDO** 命令で終了する必要があります。ENDDO により PRACTICE プログラムが終了し、現在のプログラムに関するすべての情報が PRACTICE スタックから削除されます。

ENDDO [<パラメータ>] PRACTICE プログラムから戻ります。

PRACTICE プログラムから戻るときに、結果のパラメータを渡すことができます。

```
; script test_status

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

```
; script enddo_param
DO setup_CPU

DO test_status

ENTRY &result ; Read result

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

GOSUB < ラベル > [< パラメータリスト >]	PRACTICE サブルーチンをコールします。
RETURN [< パラメータリスト >]	PRACTICE サブルーチンから戻ります。
GOTO < ラベル >	同じ PRACTICE プログラム内で分岐します。
JUMPTO < ラベル >	別の PRACTICE プログラム内のラベルに分岐します。

ENTRY <パラメータリスト>

ENTRY コマンドは以下の目的に使用できます：

- PRACTICE プログラムまたはサブルーチンにパラメータを渡す
- サブルーチンから値を返す

例 1: PRACTICE プログラムにパラメータを渡す

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

ENTRY &address_range &data_old &data_new

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

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

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

Data.Find &address_range &data_old

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

ENDDO
```

例 2: サブルーチンにパラメータを渡して戻り値を戻す

サブルーチンにパラメータを渡す

```
; Practice-program param.cmm  
  
GOSUB test sieve++0x20 0x01  
ENTRY &found ←  
PRINT "Data found at address " &found  
ENDDO  
  
test:  
→ ENTRY &address_range &data  
Data.Find &address_range &data  
&result=ADDRESS.OFFSET(TRACK.ADDRESS())  
RETURN &result
```

戻り値を戻す

論理演算の詳細については、『[IDE User's Guide](#)』 (ide_user.pdf) の「Operators」の章を参照してください。

IF/ELSE

```
IF <条件>  
  <ブロック>  
[ELSE  
  <ブロック>]
```

例:

```
; PRACTICE program patch.cmm  
  
ENTRY &address_range &data_old &data_new  
  
Data.Find &address_range &data_old  
  
IF FOUND()  
(  
    Data.Set TRACK.ADDRESS() &data_new  
    Data.Print TRACK.ADDRESS()  
    DIALOG.OK "Patch done"  
)  
ELSE  
    DIALOG.OK "Patch failed"  
ENDDO
```

ブロックの構造はCと同様です。

"(" and ") は個別のラインに記述する必要があります。

IF の後には必ずスペースが必要です。

```
Var.IF <HLL 条件>  
  <ブロック>  
[ELSE  
  <ブロック>]
```

例:

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

```
WHILE [<条件>]  
  <ブロック>  
Var.WHILE [<HLL 条件>]  
  <ブロック>  
RePeaT [<カウント>]  
  <ブロック>  
RePeaT  
  <ブロック>  
[WHILE [<条件>]]
```

PRACTICE 変数（マクロ）は、アプリケーション変数と区別するため先頭に「&」が付きます。

PRACTICE 変数はコマンドラインでは使用できないことにご注意ください。

デフォルトでは PRACTICE 変数はローカルで、変数が最初に使用されたブロック内でのみ有効です。

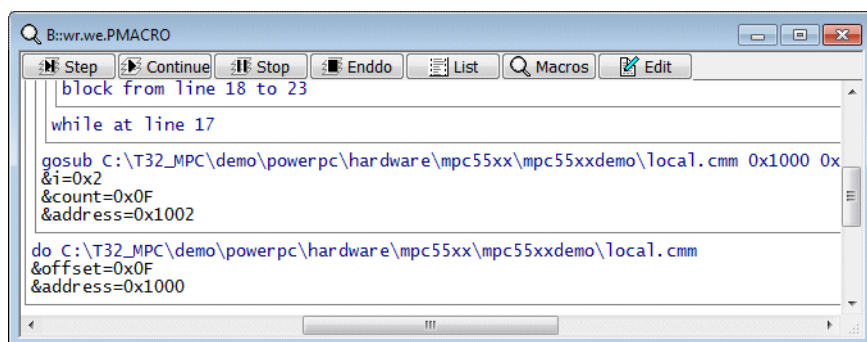
他のブロックやプログラムで PRACTICE 変数を使用するには、GLOBAL として宣言する必要があります。グローバル変数は、ENDDO または END の後も PRACTICE スタックに残ります。変数を PRACTICE スタックから削除するには、**PMACRO.RESet** を使用します。

ネストされた PRACTICE スクリプトでは、明示的に LOCAL 変数を使用することをお勧めします。

```
; script local.cmm

&address=0x1000
&offset=0xf
GOSUB display_contents &address &offset
PRINT "Contents of &address -- " &address+&offset " displayed!"
ENDDO

display_contents:
; a local variable &address for the subfunction is created
LOCAL &address
LOCAL &count
ENTRY &address &count
LOCAL &i
&i=0
WHILE &i<&count-1
(
    PRINT "Byte at &address = " DATA.BYTE(D:&address)
    WAIT 0.2s
    &i=&i+1
    &address=&address+1
)
RETURN
```



PRACTICE 変数はマクロとして、つまりテキスト置換メカニズムを介して実現されます。ある型のデータという従来の変数の概念はありません。

PRACTICE 変数には以下のものを格納できます：

変数の内容

数値

```
&i=100

GLOBAL &k
&k=&i+1

&p=V.VALUE(vint)

&f=V.VALUE(flags[4])

&float=1.4e13

&address=ADDRESS.OFFSET(flags)
```

論理演算の詳細については、『[IDE User's Guide](#)』(ide_user.pdf)の「**Operators**」の章を参照してください。

例：パラメータ化されたアドレス指定

```
; script variables1

&start_address=0xd0000000
&end_address=0xd00000ff
&new_start_address=0xd0000200

; copy data at &start_address--&end_address to &new_start_address
Data.Copy &start_address--&end_address &new_start_address

ENDDO
```

ブール値

```
&true=(0==0)
&false=(0==1)
```

```
IF RUN()
    PRINT "Program execution is running"
ELSE
    PRINT "Program execution is not running"

; script file_exists
&my_file="read.txt"

IF !OS.FILE(&my_file)
    PRINT "File ""&my_file"" does not exist"
ELSE
    TYPE &my_file
```

```
&string="String"
PRINT "&string"

&range="1000++50"
Data.dump &range
```

例: パラメータ化されたロードフォーマット

```
; script variables2

ENTRY &loadtype

IF ("&loadtype"=="")
  (
    PRINT %ERROR "No format information entered"
    ENDDO
  )

; string compares like "&loadtype"=="ELF" are case-sensitive compares

IF ("&loadtype"=="ELF")
  &format="ELF"
ELSE IF ("&loadtype"=="S19")
  &format="S3record"
ELSE
  (
    PRINT %ERROR "No valid format"
    ENDDO
  )

Data.LOAD.&format *

ENDDO
```

複雑なデータ構造ではすべて擬似変数が使用されます。

擬似変数は **Var.NEW** <型> ¥<変数名> で宣言し、アプリケーションプログラム変数のように PRACTICE プログラムで使用できます。

例 1: 配列

```
Var.NEW char[10] ¥a1
Var.Set ¥a1[4]=1
```

例 2: 構造体

```
struct abc
{
    int x;
    int y;
}
```

```
Var.NEW struct abc ¥abc
Var.Set ¥abc.x=0
```

各 PRACTICE コマンドは以下の 3 ステップで実行されます：

1. すべてのマクロをマクロに含まれる文字シーケンスと置き換える
2. すべての式を評価する
3. コマンドを実行する

例 1

```
&a="3+4"  
PRINT &a
```

PRINT &a コマンドの実行

1. マクロを文字シーケンス PRINT 3+4 と置き換えます。
2. 式 PRINT 7 を評価します。
3. コマンドを実行します。

例 2

```
&a="Hello World" ; not working version  
PRINT &a
```

PRINT &a コマンドの実行

1. マクロを文字シーケンス PRINT Hello World と置き換えます。
2. 式 Hello World の評価が失敗し、**Symbol not found** というエラーメッセージが返されます。
TRACE32 は *Hello* と *World* をプログラムシンボルとして解釈するため、スクリプトは動作しません。

例 3

```
&a="Hello World" ; correct version  
PRINT "&a"
```

PRINT "&a" コマンドの実行

1. マクロを文字シーケンス PRINT "Hello World" と置き換えます。
2. 式を評価します（何もしない）。
3. コマンドを実行します。

例 4

```
&a="Hello World" ; not working version
&b=&a
```

&b= コマンドの実行

1. マクロを文字シーケンス `&b=Hello World` と置き換えます。
2. 式 `Hello World` の評価が失敗し、**Symbol not found** というエラーメッセージが返されます。
TRACE32 は *Hello* と *World* をプログラムシンボルとして解釈するため、スクリプトは動作しません。

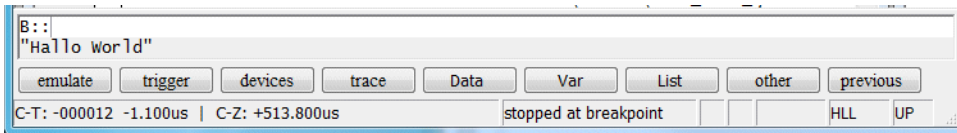
```
&a="Hello World" ; correct version
&b="&a"
```

例 5

```
; script Return_String1.cmm

GOSUB ReturnWithString
ENTRY &a ; &a="Hello World" (with quote signs included)
PRINT "&a"
ENDDO

ReturnWithString:
LOCAL &b
&b="Hallo World"
RETURN "&b"
```



出力する場合は、&a="Hello World" に大きな問題はないように思われるでしょうが、マクロを他の目的に使用する場合は問題です。以下に解決方法を示します：

```
GOSUB ReturnWithString
ENTRY %LINE &a ; %LINE copies the return value without evaluation
PRINT "&a"
ENDDO

ReturnWithString:
LOCAL &b
&b="Hallo World"
RETURN &b
```

16 進数の 10 進数への変換

以下の例では、次の TRACE32 関数を使用しています：

V.VALUE(< 式 >)

HLL 式の内容を 16 進数で返します。

FORMAT.DECIMAL(< 幅 >, < 数字 >)

数値式を固定幅の 10 進数にフォーマットします。

```
PRINT %Decimal V.VALUE(ast.count) ; use decimal format for printing
```

```
PRINT V.VALUE(ast.count)+0. ; type conversion by adding a  
; decimal number
```

```
; formatting to a fixed width decimal number  
PRINT FORMAT.DECIMAL(0,V.VALUE(ast.count))
```

出力された値を見る人が 10 進数だと明確に判断できるように、「.」を付けることもできます。

```
PRINT %Decimal V.VALUE(ast.count) "."
```

文字列の演算

```
; script Return_String1.cmm  
  
&r=STRING.MID(SOFTWARE.VERSION(),0,1)  
IF "&r"=="R"  
(  
    &v=STRING.MID(SOFTWARE.VERSION(),11.,9.)  
    WHILE STRING.FIND("&v","0")  
        &v=STRING.CUT("&v",1)  
    PRINT "Release "+"&r"+"&v"  
)  
ELSE  
    PRINT "This is not an offical release"  
ENDDO
```

出カコマンド

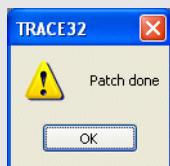
PRINT <フォーマット><パラメータリスト>

DIALOG.OK <メッセージ>

```
PRINT "FLASH programmed successfully"
```

```
PRINT %ERROR "FLASH programming failed"
```

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



ENTER <パラメータリスト>	ウィンドウベースの入力です。
INKEY [<パラメータ>]	入コマンド（文字）です。
DIALOG.YESNO <メッセージ>	標準ダイアログを作成します。
DIALOG.File <メッセージ>	ダイアログを介してファイル名を読み取ります。

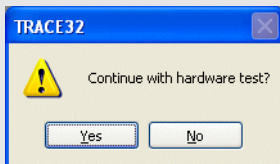
```

INKEY                                ; Wait for any key

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

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

```



```

ENTRY &result

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

```

```

DIALOG.File *sre
ENTRY &filename

Data.LOAD.S3record &filename

ENDDO

```

PRACTICE の入出力に I/O ウィンドウが必要です。そのために [AREA] ウィンドウを使用します。

[AREA] ウィンドウを開いて割り当てる

1. [AREA] ウィンドウを作成します。

```
AREA.Create [< 領域 >]
```

2. PRACTICE の I/O のために [AREA] ウィンドウを選択します。

```
AREA.Select [< 領域 >]
```

3. [AREA] ウィンドウの画面上の位置を選択します。[AREA] ウィンドウに名前を割り当てることのできるため、ここでこのコマンドを使用します。これは、I/O プロシージャの後でこのウィンドウを削除したい場合に便利です。

```
WinPOS [< 位置 >] [< サイズ >] [< スケール >] [< 名前 >] [< 状態 >] [< ヘッダー >]
```

4. [AREA] ウィンドウを表示します。

```
AREA.view [< 領域 >]
```

[AREA] ウィンドウの削除

1. [AREA] ウィンドウの設定をデフォルト設定にリセットします：メッセージエリア（AREA A000）がエラーメッセージとシステムメッセージに使用されます。これ以外の [AREA] ウィンドウはアクティブではありません。

```
AREA.RESet
```

2. 特定のウィンドウを削除します。

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

```
; PRACTICE file iowindow.cmm

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

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

WAIT 2.s

AREA.RESet
WinCLEAR IO1

ENDDO
```

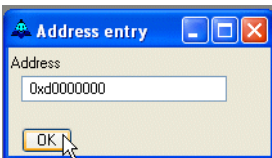
DIALOG プログラミング機能を用いて I/O を実現することもできます。

```
; PRACTICE file dialog_prog.cmm

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

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

DIALOG.END
ENDDO
```



STOP

PRACTICE プログラムの実行を停止します。

上記の例では、DEFBUTTON に割り当てられる GOTO okclose コマンドにより、PRACTICE プログラムの実行が続行します。

```

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

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

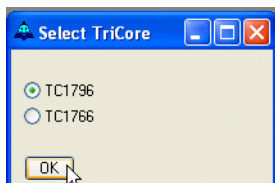
    STOP
    IF DIALOG.BOOLEAN(TC.1796)             ; Evaluate result
        CPU="TC1796"

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

DIALOG.END

PRINT "Selected CPU= " "&CPU"
ENDDO

```



DIALOG プログラミングコマンドの詳細については、『*Operation System Reference*』を参照してください。

ファイルコマンド

ファイルコマンドを使用して、データをファイルに保存したり、ファイルからテストデータを読み取ります。

- ファイルを開く

```
OPEN #< バッファ>< ファイル名> / Read | Write | Create
```

- ファイルを閉じる

```
CLOSE #< バッファ>
```

- 開いたファイルからデータを読み取る

```
READ #< バッファ> [%LINE] <パラメータリスト>
```

- 開いたファイルにデータを書き込む

```
WRITE #< バッファ><パラメータリスト>
```

例 1:

```
; PRACTICE file write.cmm

OPEN #1 abc.txt /Create

&i=0

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

CLOSE #1

ENDDO
```

例 2: ファイルからデータを読み取ります。ファイル内の個々のデータは空白またはキャリッジリターンで区切られます。

```
; PRACTICE file read.cmm

OPEN #2 read.txt /Read

READ #2 &e1 &e2 &e3 &e4

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

CLOSE #2

ENDDO
```

例 3: ファイルからデータを読み取ります。常に完全なラインを読み取ります。

```
; PRACTICE file read_line.cmm

OPEN #2 read.txt /Read

READ #2 %LINE &e1

PRINT "&e1"

CLOSE #2

ENDDO
```

- ON ERROR < コマンド >** PRACTICE 実行時エラーが発生したときにコマンド実行します。
- ON SYSUP < コマンド >** デバッガと CPU の通信が確立したときにコマンドを実行します。
- ON POWERUP < コマンド >** ターゲットの電源がオンになったとき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
```

