


TRACE32 Online Help	
TRACE32 Directory	
TRACE32 Index	
TRACE32 Training .....	
HLL Debugging - トレーニング .....	2
アプリケーションプログラムのロード .....	3
シンボルデータベース .....	12
内部シンボルデータベースの構造 .....	12
シンボルデータベースの一般情報 .....	13
シンボルブラウザ .....	14
選択したシンボルの詳細 .....	20
ソースファイル内の検索 .....	22
変数の表示 .....	24
[Watch] ウィンドウ .....	24
[View] ウィンドウ .....	26
参照される変数 .....	27
ローカル変数 .....	28
スタックフレーム .....	29
配列用の特殊な表示 .....	30
リンクしたリスト .....	33
変数ログ .....	35
変数のフォーマット .....	39
[Format] ダイアログボックスを使用した変数のフォーマット .....	39
コマンドラインを使用した変数のフォーマット .....	49
一般的な SETUP .....	50
関数のテスト .....	51
索引 (ローカル) .....	54

08/19/04 「アプリケーションプログラムのロード」セクションを改定。

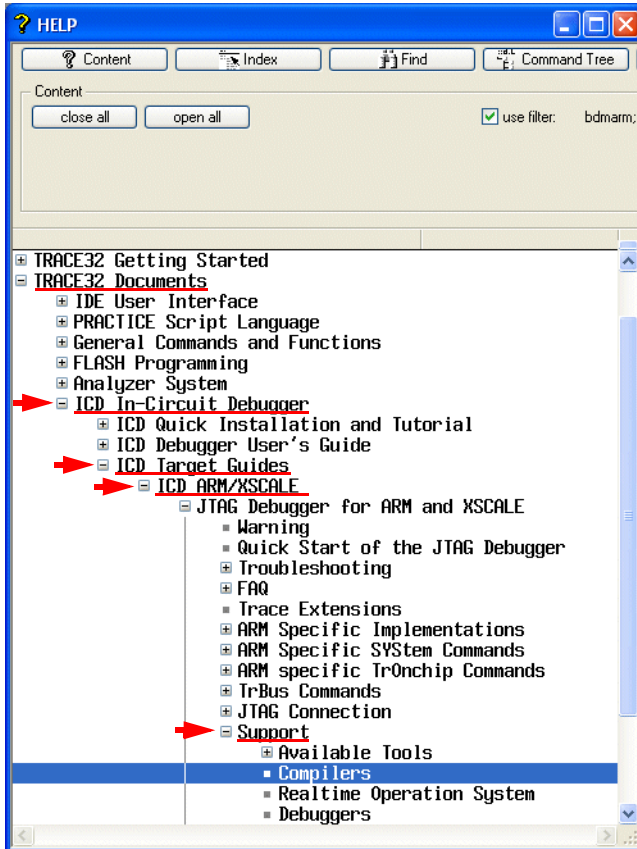
**Data.LOAD** < ファイル名 > [/< オプション >]

汎用ロードコマンド

**Data.LOAD.<サブコマンド>** < ファイル名 > [/< オプション >]

コンパイラ固有のロードコマンド

コンパイラ固有のロードコマンドは以下の場所にあります：



コンパイラ/フォーマット固有のすべてのオプションを使用できるように、コンパイラ/フォーマット固有の **Data.LOAD** コマンドを使用することをお勧めします。

- TRACE32 ですでに使用可能なすべてのシンボルとデバッグ情報が削除される。
- <ファイル> が提供するコード / データがターゲットメモリにロードされる。
- <ファイル> が提供するシンボルおよびデバッグ情報が TRACE32 にロードされる。
- <ファイル> が提供する HLL ソースファイルのパスが TRACE32 にロードされる。
- ロードされた情報から内部シンボルデータベースが生成される。

## コード/データを参照するオプション

コードやデータを参照するオプションは主に以下のタスクに使用されます：

- コード/データが正しくロードされたことを検証する。
- 正しいコード/データがすでにターゲットに存在する場合は、コード/データのロードを抑制する。これは FLASH プログラミング時間を節約する場合などに役立つ。

<b>Verify</b>	書き込み後にメモリを検証します。そのために、メモリブロックに書き込んだ直後にメモリをリードバックします。
<b>ComPare</b>	メモリのデータをファイルと比較します。メモリは変更されません。1 つ目の差異が検出されると比較が停止します。
<b>DIFF</b>	メモリのデータをファイルと比較します。メモリは変更されません。 <ul style="list-style-type: none"><li>• ファイルとメモリに差異が検出された場合は、FOUND() が TRUE を返す。</li><li>• ファイルとメモリに差異が検出されない場合は、FOUND() が FALSE を返す。</li></ul>
<b>NOCODE</b>	シンボルとデバッグ情報および HLL ソースパス情報のみをダウンロードします。コード/データがすでにメモリに置かれている場合に便利です。

```
Data.LOAD.Elf armle.axf /StripPATH /LowerPATH ; load ELF file for ARM
Data.LOAD.AIF armla.axf /StripPATH /LowerPATH ; load AIF file for ARM
Data.LOAD.COFF arm.abs ; load COFF file for ARM
Data.LOAD.COFF arm.abs /DIFF ; reprogram the FLASH
IF FOUND() ; only if the code
( ; changed, otherwise
    FLASH.Program ALL ; load only symbol and
    Data.LOAD.COFF arm.abs ; debug information plus
    FLASH.Program OFF ; hll source path
) ; information
ELSE
    Data.LOAD.COFF arm.abs /NOCODE
```

## シンボルおよびデバッグ情報を参照するオプション

シンボルおよびデバッグ情報を参照するオプションは、主にシンボル情報を再配置するために使用します。

```
; relocate all symbols by 2000
symbol.RELOCate.shift 2000

; Load the symbol and debug information from the file t_li_elf.axf and
; relocate all symbols of the section t_li_elf.axf to address 2000
sYmbol.List.SECTION
Data.LOAD.Elf thumble.axf /RELOC t_li_elf.axf AT 2000 /NOCODE
```

**sYmbol.RELOCate.shift** < オフセット >

< オフセット > の値でコードおよびデータシンボルを再配置します。

**Data.LOAD.Elf** < ファイル > /**RELOC** < セクター > **AT** < アドレス >

指定したセクターを定義したアドレスに再配置します。

**Data.LOAD.Elf** < ファイル > /**RELOC** < セクター > **AFTER** < セクター\_2 >

指定したセクターを別のセクターの後ろに再配置します。

**sYmbol.List.SECTION**

< ファイル > が提供するセクション情報をリストします。

<b>NoClear</b>	デフォルトでは、新しい <b>Data.LOAD</b> コマンドが開始されるたびに、すでに使用可能なシンボルおよびデバッグ情報が削除されます。このオプションを指定すると、すでに使用可能なシンボルおよびデバッグ情報は削除されません。このオプションは複数のプログラムをロードする場合に必要です。
<b>MultiFile</b>	このオプションで、複数のプログラムを含む大規模プロジェクトのダウンロードを高速化します。このオプションを指定すると、 <b>Data.LOAD</b> コマンドを使用したときの内部シンボルデータベースの生成が抑制されます。

```
Data.LOAD file1 /MultiFile           ; load file1 but suppress the
                                       ; generation of the internal
                                       ; symbol data base

Data.LOAD file2 /NoClear /MultiFile   ; load file2 but don't remove the
                                       ; already available symbol and
                                       ; debug information before
                                       ; loading and suppress the
                                       ; generation of the internal
                                       ; symbol data base

Data.LOAD file3 /NoClear /MultiFile

.
.
.

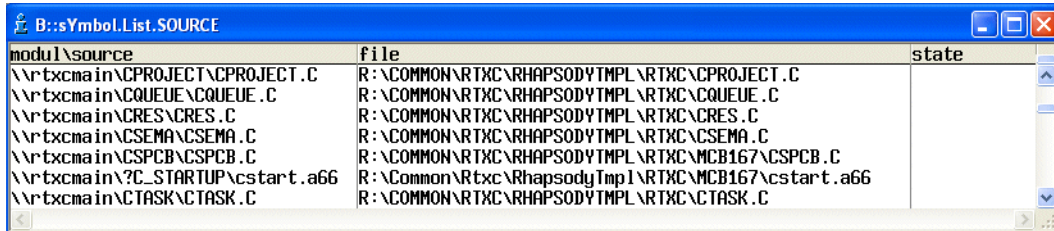
Data.LOAD filen /NoClear              ; load filen but don't remove the
                                       ; already available symbol and
                                       ; debug information before
                                       ; loading, this is the last file
                                       ; so generate the internal symbol
                                       ; data base now
```

## HLL ソースファイルの正しいパスを取得するオプションとコマンド

**sYmbol.List.SOURCE** コマンドを使用して、<ファイル> からロードされた HLL ソースファイルのパス情報を表示します。

### ベースディレクトリの修正

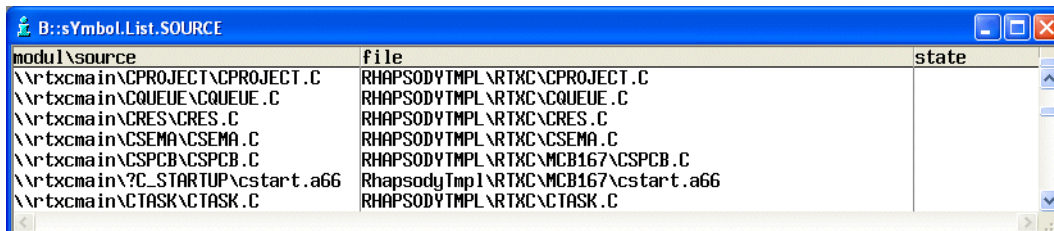
<b>StripPART &lt; 数字 &gt;</b>	所定のパスの最初の n 個部分を削除します。
-------------------------------	------------------------



modul\source	file	state
\\rtxcmain\CPROJECT\CPROJECT.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\CPROJECT.C	
\\rtxcmain\CQUEUE\CQUEUE.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\CQUEUE.C	
\\rtxcmain\CRES\CRES.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\CRES.C	
\\rtxcmain\CSEMA\CSEMA.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\CSEMA.C	
\\rtxcmain\CSPCB\CSPCB.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\MCB167\CSPCB.C	
\\rtxcmain\?C_STARTUP\cstart.a66	R:\Common\Rtxc\RhapsodyTmpl\RTXC\MCB167\cstart.a66	
\\rtxcmain\CTASK\CTASK.C	R:\COMMON\RTXC\RhapsodyTmpl\RTXC\CTASK.C	

パスのほとんどの部分は正しいですが、HLL ソースファイルはベースディレクトリ  
G:¥AND¥F167¥Common¥RTXC に置かれ、R:¥Common¥RTXC には置かれていません。

```
Data.LOAD.Omf rtxcmain /StripPART 3. ; the option StripPART 3.  
; removes the first 3  
; parts from the path
```



modul\source	file	state
\\rtxcmain\CPROJECT\CPROJECT.C	RHAPSODYTMPL\RTXC\CPROJECT.C	
\\rtxcmain\CQUEUE\CQUEUE.C	RHAPSODYTMPL\RTXC\CQUEUE.C	
\\rtxcmain\CRES\CRES.C	RHAPSODYTMPL\RTXC\CRES.C	
\\rtxcmain\CSEMA\CSEMA.C	RHAPSODYTMPL\RTXC\CSEMA.C	
\\rtxcmain\CSPCB\CSPCB.C	RHAPSODYTMPL\RTXC\MCB167\CSPCB.C	
\\rtxcmain\?C_STARTUP\cstart.a66	RhapsodyTmpl\RTXC\MCB167\cstart.a66	
\\rtxcmain\CTASK\CTASK.C	RHAPSODYTMPL\RTXC\CTASK.C	

```
; define the new directory as base for relative paths  
sYmbol.SourcePATH.SetBaseDir G:¥AND¥F167¥Common¥RTXC
```

modul\source	file	state
\\r\txcmain\CPROJECT\CPROJECT.C	RHAPSODYTMPL\RTXC\CPROJECT.C	
\\r\txcmain\CQUEUE\CQUEUE.C	G:\AND\F167\Common\RTXC\RHAPSODYTMPL\RTXC\CQUEUE.C	loaded
\\r\txcmain\CRES\CRES.C	RHAPSODYTMPL\RTXC\CRES.C	
\\r\txcmain\CSEMA\CSEMA.C	G:\AND\F167\Common\RTXC\RHAPSODYTMPL\RTXC\CSEMA.C	loaded
\\r\txcmain\CSPCB\CSPCB.C	RHAPSODYTMPL\RTXC\MCB167\CSPCB.C	
\\r\txcmain\C_STARTUP\cstart.a66	G:\AND\F167\Common\RTXC\RhapsodyTmp1\RTXC\MCB167\cstart.a66	loaded
\\r\txcmain\CTASK\CTASK.C	RHAPSODYTMPL\RTXC\CTASK.C	

dir	base	cach	rec	dun	hit	directory
	√				√	G:\AND\F167\Common\RTXC

### sYmbol.List.SOURCE

HLL ソースファイルのパス情報を表示します。

**Data.Load** < ファイル > /StripPART < 数字 >

< ファイル > からロードされたパスの最初の n 個部分を削除します。

**sYmbol.SourcPATH.SetBaseDir** < ディレクトリ >

ベースとして新しいベースディレクトリを相対パスで定義します。

**sYmbol.SourcePATH.List**

ソースファイルのすべての検索ディレクトリを表示します。

**sYmbol.List.Source** でリストされたパスから HLL ソースファイルをロードできない場合、TRACE32 は定義された検索ディレクトリで HLL ソースファイルの検索を試みます。

以下のコマンドを使用して新しい検索パスを定義できます：

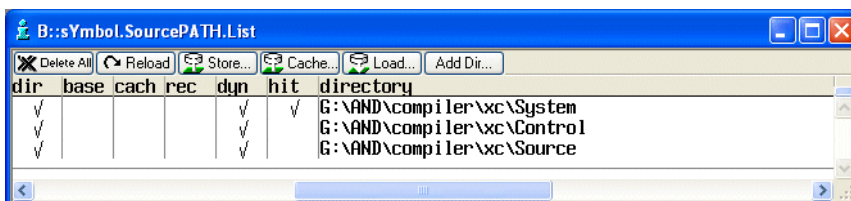
<b>sYmbol.SourcePATH.SetDir</b> <ディレクトリ>	直接検索パスを定義します（静的）。
<b>sYmbol.SourcePATH.SetDynamicDir</b> <ディレクトリ>	直接検索パスを定義します（動的）。
<b>sYmbol.SourcePATH.SetRecurseDir</b> <ディレクトリ>	再帰的検索パスを定義します。定義したディレクトリとそのすべてのサブディレクトリが検索パスとして使用されます。

静的な直接パスを定義した場合は、検索パスが常に同じ順序でスキャンされます。

```
sYmbol.SourcePATH.SetDir G:¥AND¥compiler¥xc¥Control
sYmbol.SourcePATH.SetDir G:¥AND¥compiler¥xc¥Source
sYmbol.SourcePATH.SetDir G:¥AND¥compiler¥xc¥System
```

動的な直接検索パスを定義した場合は、最後に検索したソースファイルが置かれていたディレクトリが、次のソースファイルを検索する最初のディレクトリになります。

```
sYmbol.SourcePATH.SetDynamicDir G:¥AND¥compiler¥xc¥Control
sYmbol.SourcePATH.SetDynamicDir G:¥AND¥compiler¥xc¥Source
sYmbol.SourcePATH.SetDynamicDir G:¥AND¥compiler¥xc¥System
Data.List ad_cond
```



## 仮想メモリのローダーオプション

TRACE32 はホストにいわゆる仮想メモリを提供します。以下のオプションで、コードをこの仮想メモリにロードします。

<b>VM</b>	コード / データを仮想メモリにロードします。
<b>PlusVM</b>	コード / データをターゲットおよび仮想メモリにロードします。

```
Data.LOAD.COFF arm.abs /VM           ; load code/data from <file> into the
                                       ; virtual memory

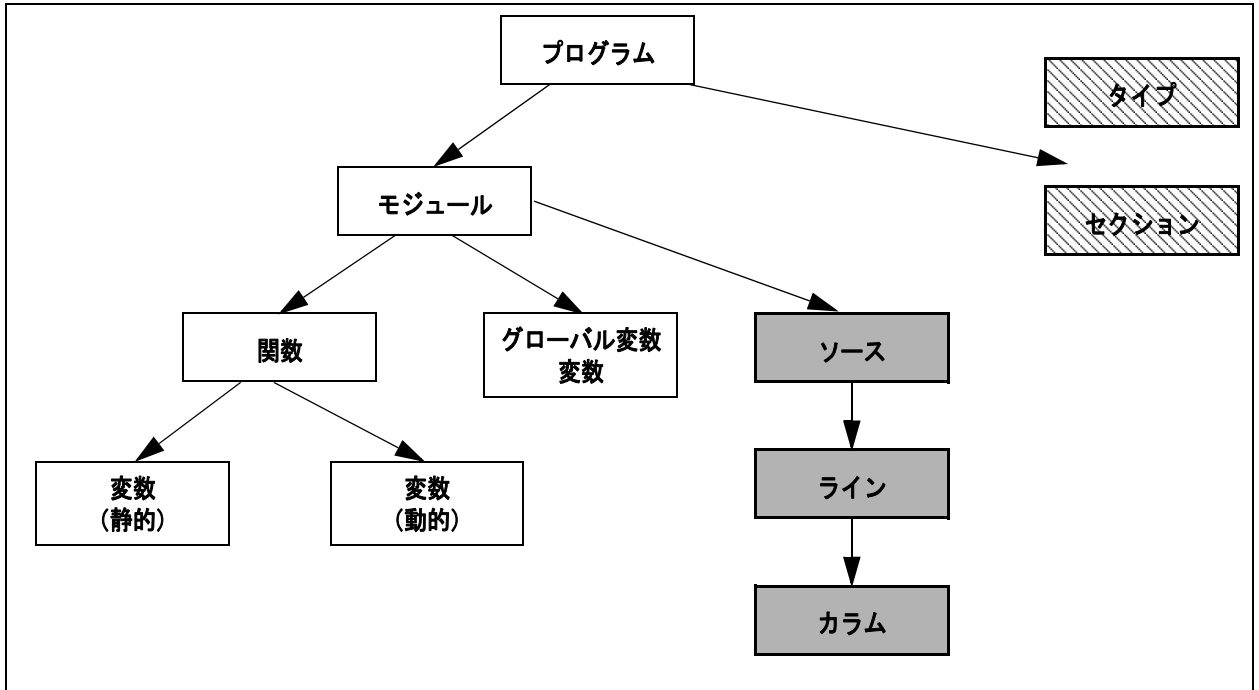
Data.LOAD.COFF arm.abs /PlusVM        ; load code data from <file> into the
                                       ; target memory and into the virtual
                                       ; memory
```

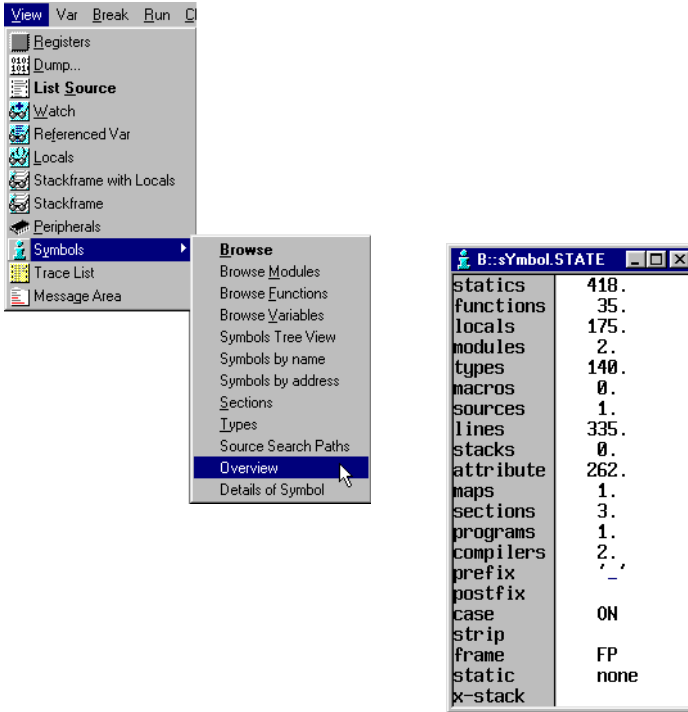
仮想メモリは主に、MPC500/800、ARM-ETM などのプログラムフロートレースに使用されます。削減されたトレース情報だけがサンプリングされるため、完全なトレースリストを表示するために TRACE32 はターゲットメモリのコードも必要とします。プロセッサのオンチップデバッグロジックが、プログラム実行中のメモリリードをサポートしない場合、プログラム実行を停止しなければ、完全なトレースを表示できません。

コードが仮想メモリにロードされる場合は、TRACE32 が仮想メモリのコードを使用して完全なトレースリストを表示できます。

## 内部シンボルデータベースの構造

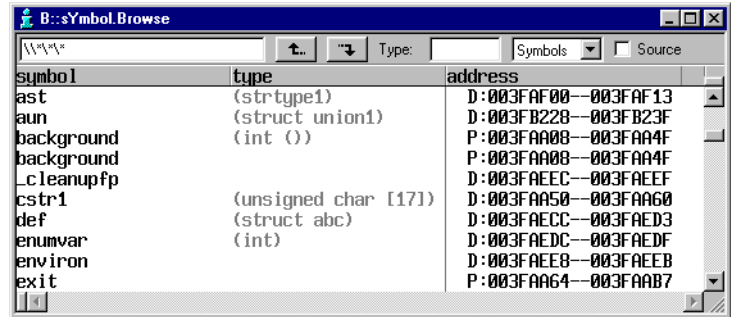
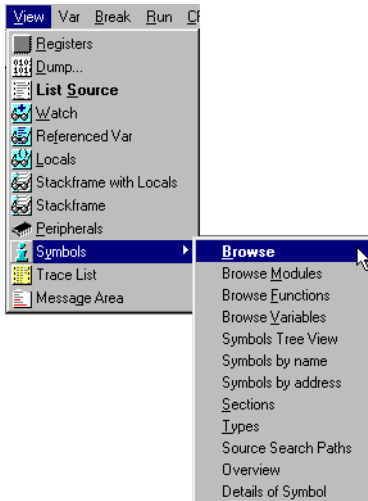
Data.LOAD コマンドでロードされたシンボルおよびデバッグ情報は、TRACE32 によって内部シンボルデータベースに整理されます。





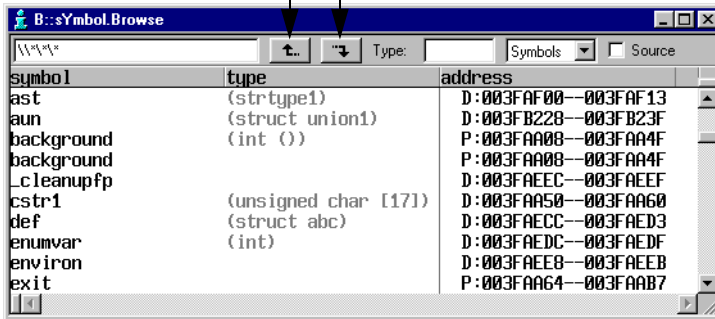
## sYmbol.STATE

シンボルデータベースについての一般情報を表示します。



**sYmbol.Browse** [<名前パターン>] [<型パターン>] [<オプション>] シンボル情報をブラウザします。  
シオン>]

グローバルに上へ    グローバルに下へ

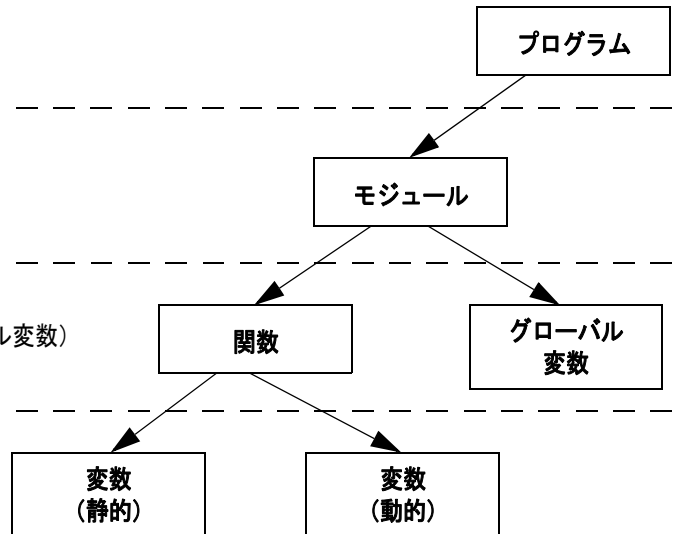


**\*\***\* (すべてのプログラム)

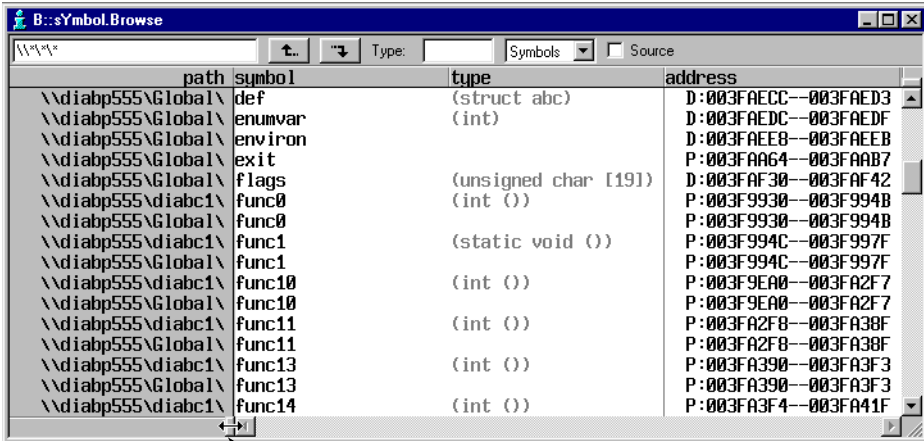
**\*\*****\*\***\* (すべてのモジュール)

**\*\*****\*\*****\*\***\* (すべての関数、すべてのグローバル変数)

**\*\*****\*\*****\*\*****\*\***\* (すべてのローカル変数)

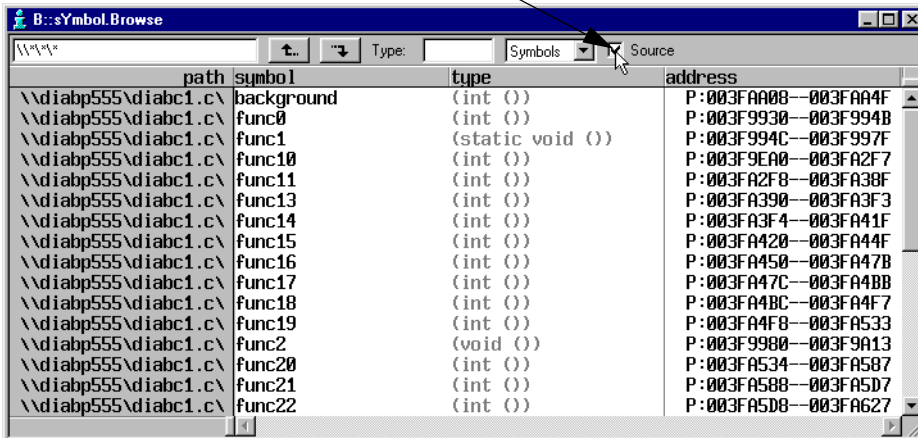


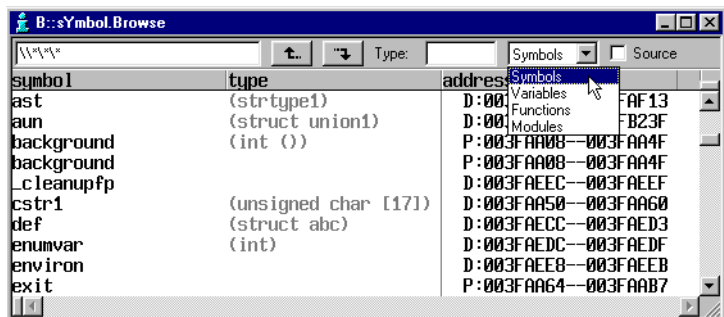
## モジュール名の代わりにファイル名を表示



シンボルのモジュール名を表示するには、小さな四角形を右に移動します。

ソースがオンの場合は、モジュール名の代わりにソースファイルの名前が表示されます。





表示タイプの選択	
[Symbols]	すべてのシンボルを表示します。
[Variables]	すべての変数を表示します。
[Functions]	すべての関数を表示します。
[Modules]	すべてのモジュールを表示します。

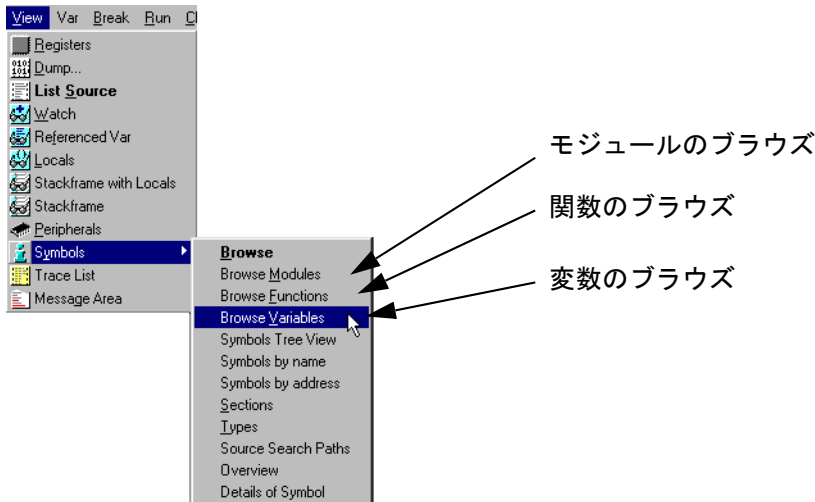
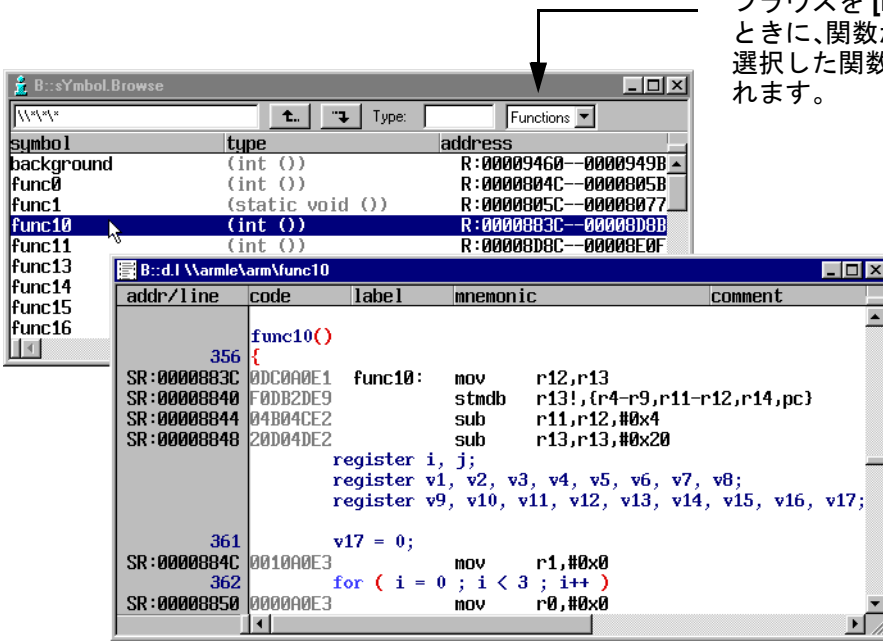
ブラウザを [Variables] に絞り込んだときに、関数が選択されている場合は、選択した関数のローカル変数が表示されます。

symbol	type	address
def	(struct abc)	D:0000E138--0000E13F
enumvar	(enumtyp)	D:0000E148--0000E148
flags	(unsigned char [19..	D:0000FAC8--0000FADA
func0		
func1		
func10		
func11		
func13		

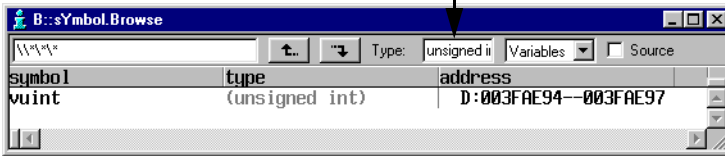
symbol	type	address
i	(auto int)	R11+FFFFFFD8--FFFFFFDB
j	(auto int)	R11+FFFFFFD4--FFFFFFD7
return	(register int)	R0
v1	(register int)	R9
v10	(register int)	R2
v11	(auto int)	R11+FFFFFFD0--FFFFFFD3
v12	(auto int)	R11+FFFFFFC0--FFFFFFCF
v13	(auto int)	R11+FFFFFFC8--FFFFFFCB
v14	(auto int)	R11+FFFFFFC4--FFFFFFC7
v15	(auto int)	R11+FFFFFFC0--FFFFFFC3
v16	(auto int)	R11+FFFFFFBC--FFFFFFBF
v17	(register int)	R1
v2	(register int)	R8
v3	(register int)	R7
v4	(register int)	R6
v5	(register int)	R5
v6	(register int)	R4
v7	(register int)	R14
v8	(register int)	R12
v9	(register int)	R3

ブラウザを [Functions] に絞り込んだときに、関数が選択されている場合は、選択した関数のソースコードが表示されます。

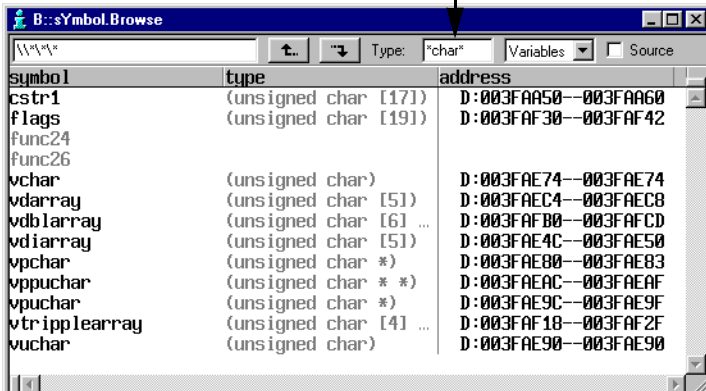


- sYmbol.Browse.sYmbol** シンボル情報をブラウザします。
- sYmbol.Browse.Function** 関数をブラウザします。
- sYmbol.Browse.Var** 変数をブラウザします。
- sYmbol.Browse.Modules** モジュールをブラウザします。

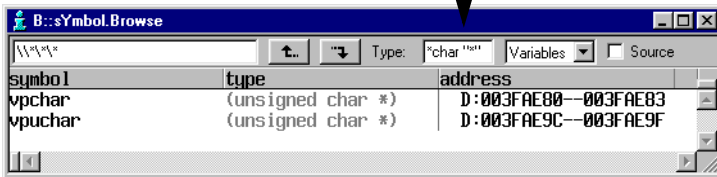
[unsigned int] 型のすべての変数を表示します。



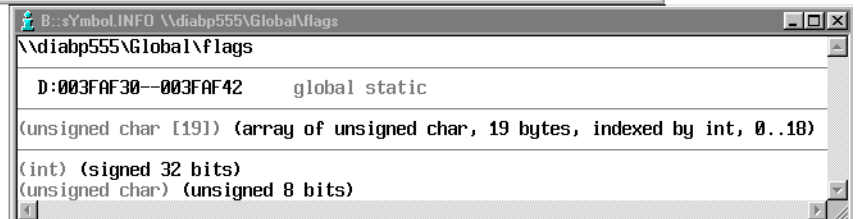
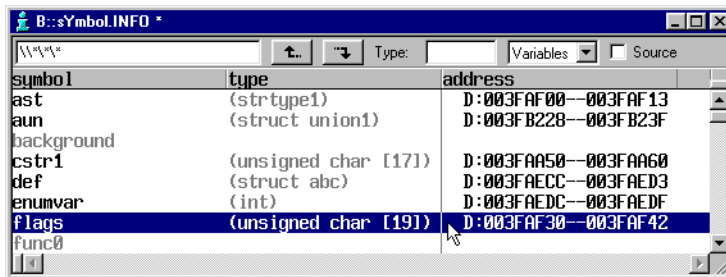
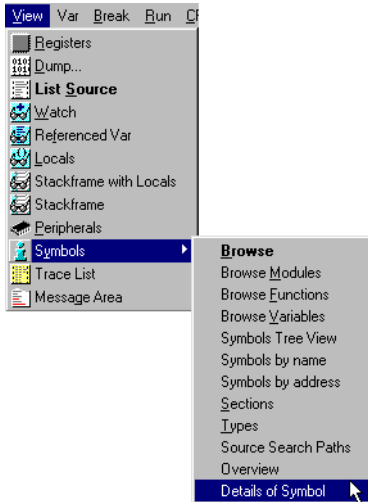
型名にキーワード char が含まれるすべての変数を表示します ([\*char\*])。

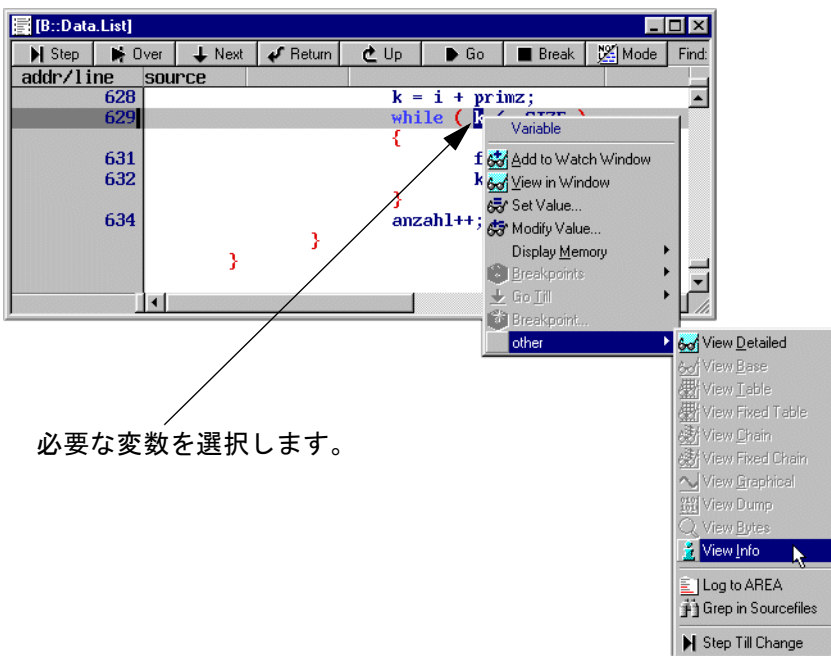


char へのポインタ型のすべての変数を表示します ([\*char "\*" ] )。



# 選択したシンボルの詳細





必要な変数を選択します。



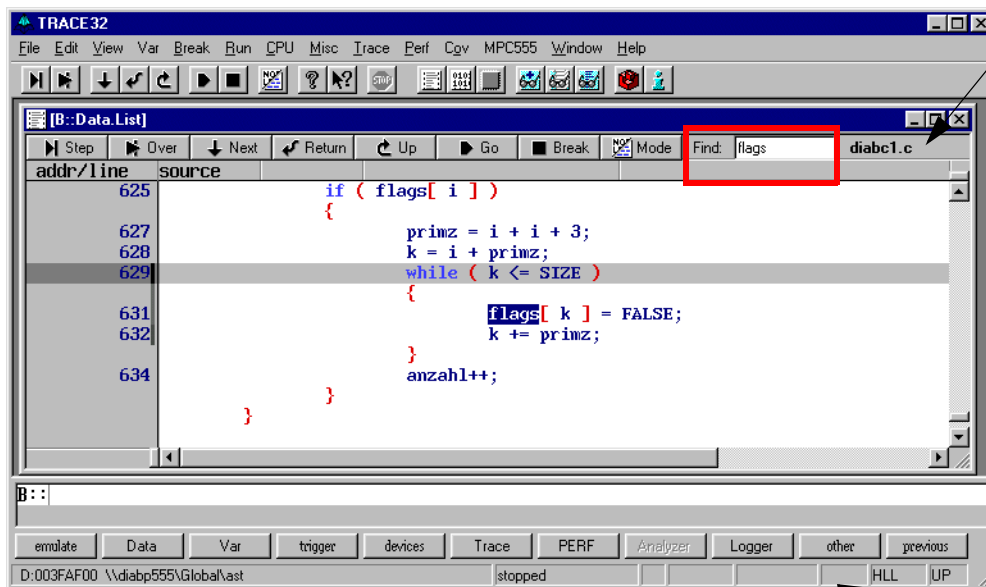
## sYMBOL.INFO

シンボリックアドレス、シンボルの場所、範囲、レイアウトを表示します。

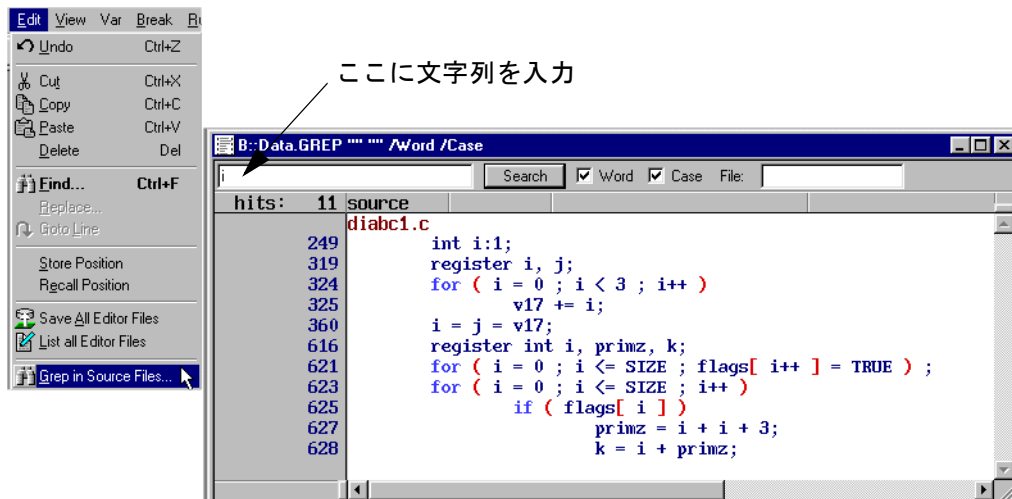
## 現在のソースファイルの文字列の検索

[HLL] デバッグモードがアクティブな場合は、入力した文字列が現在のソースファイル内で検索されます。

現在のソースファイル



[HLL] デバッグモードがアクティブ

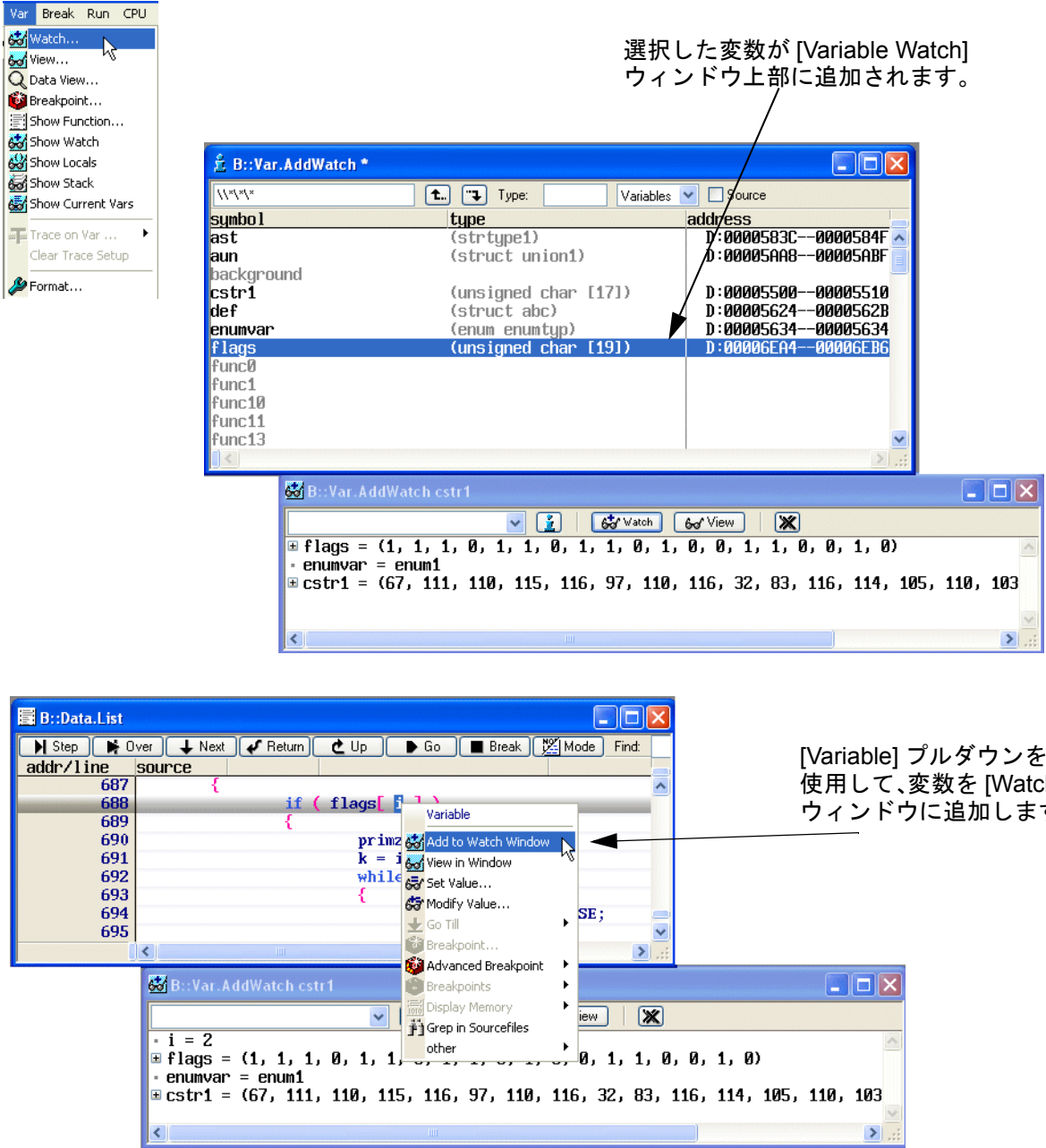


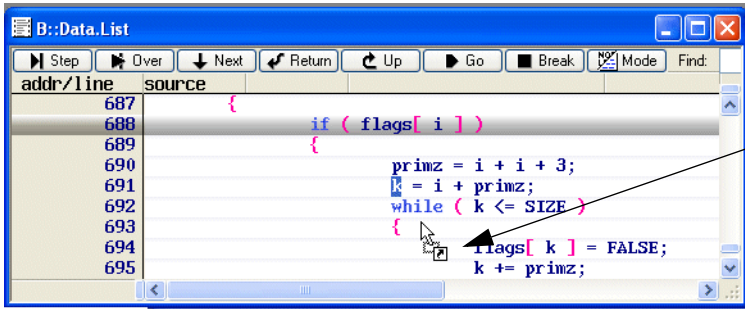
ここに文字列を入力

TRACE32 が定義した文字列をすべてのソースファイル内で検索します。

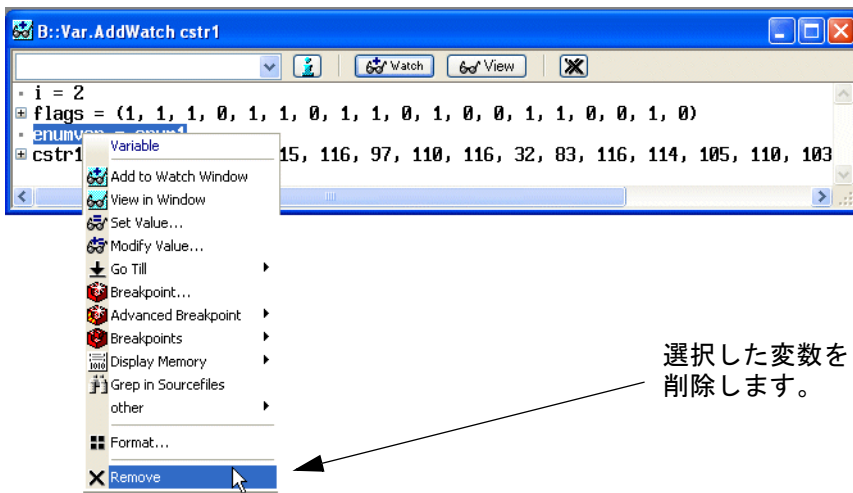
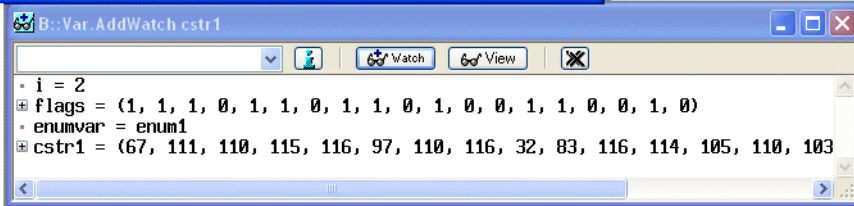
## [Watch] ウィンドウ

選択した変数を **[Variable Watch]** ウィンドウ上部に追加します。[Watch] ウィンドウがない場合は新しい [Watch] ウィンドウが作成されます。





変数を [Watch] ウィンドウにドラッグします。



選択した変数を [Watch] ウィンドウから削除します。

**Var.Watch** [<% フォーマット >] [< 変数 >]

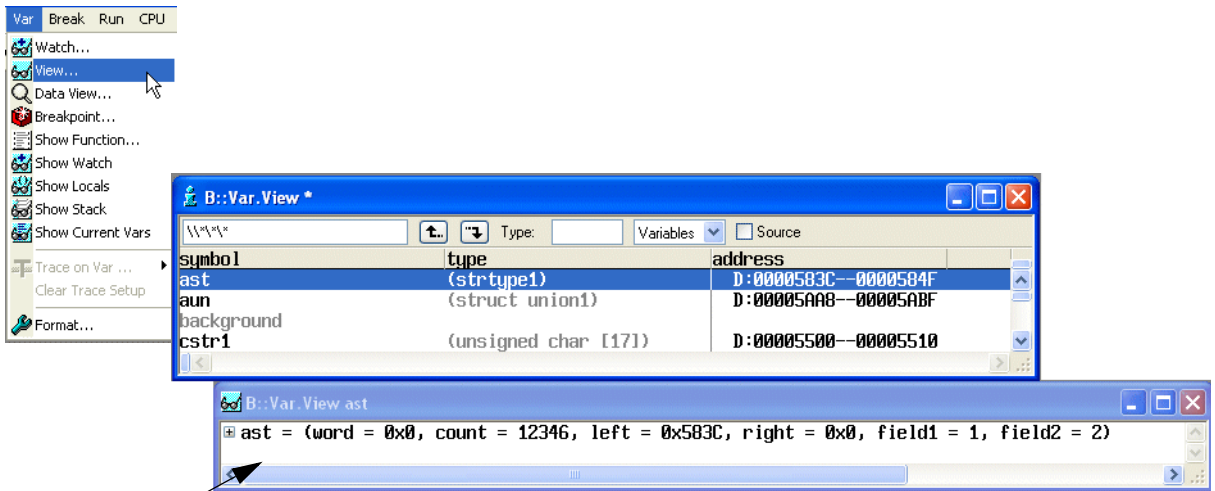
[Watch] ウィンドウを開いて変数を表示します。

**Var.AddWatch** [<% フォーマット >] [< 変数 >]

[Watch] ウィンドウに変数を追加します。

## [View] ウィンドウ

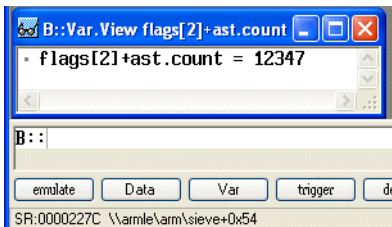
選択した変数用の新しい **[Variable View]** ウィンドウを開きます。



新しい **[Variable View]** ウィンドウが開いて  
選択した変数が表示されます。

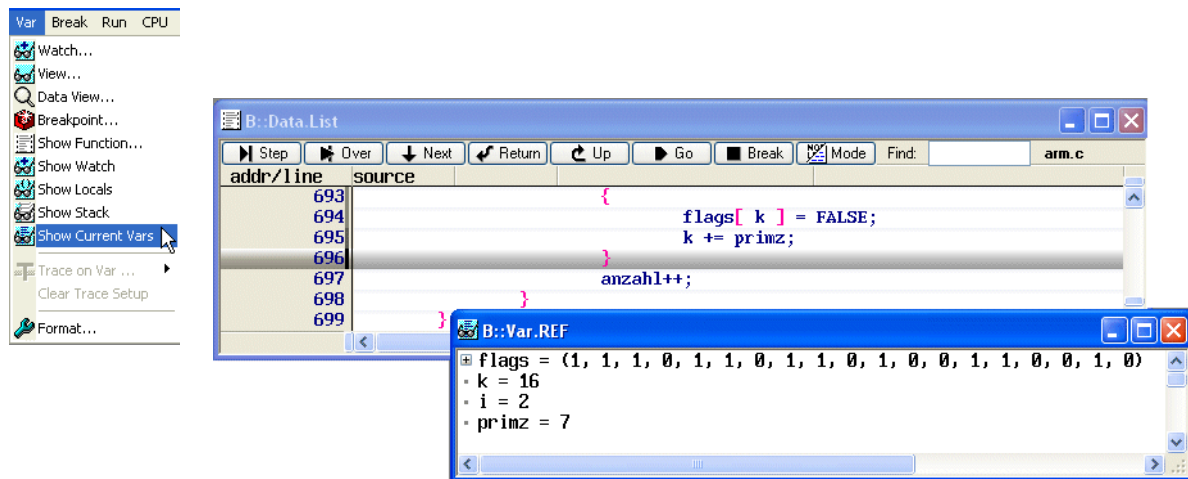
**Var.View** [ $\%$  フォーマット>] [< 変数 >]      個別のウィンドウに変数を表示します。

- 式を入力すると、その式が処理されて結果が表示されます。



## 参照される変数

[Var.REF] ウィンドウを開きます。現在のソースラインが参照する変数が自動的にこのウィンドウに追加されます。

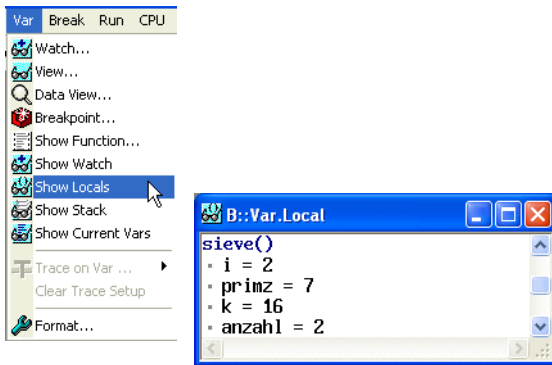


**Var.Ref** [<% フォーマット>] [/TRACK]

現在のコードラインが参照する変数を表示します。

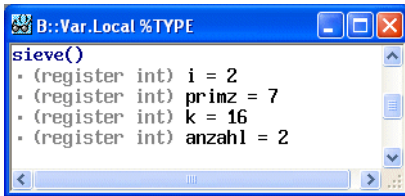
# ローカル変数

ウィンドウを開いて、現在の関数のローカル変数を表示します。



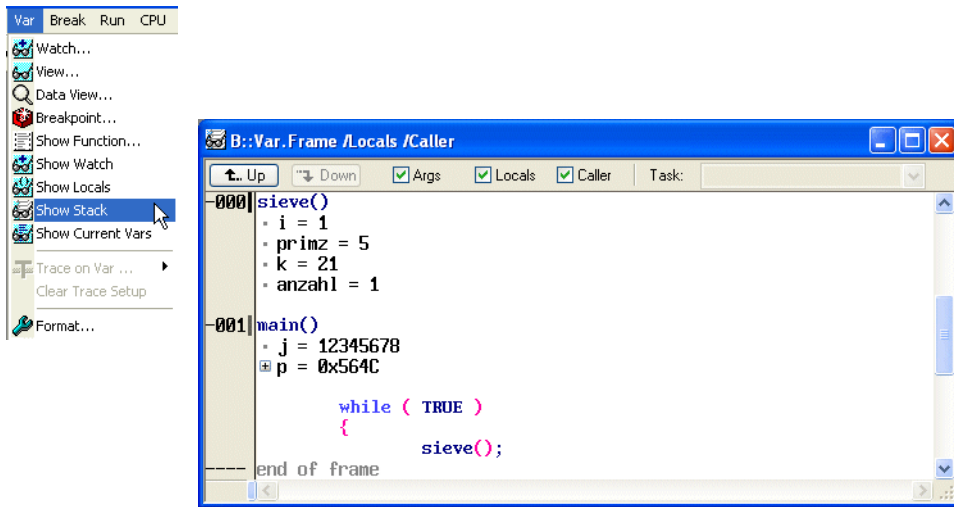
**Var.Local** [<% フォーマット>]

ローカル変数を表示します。



# スタックフレーム

スタックトレースを表示して、関数のネスト状態を示します。



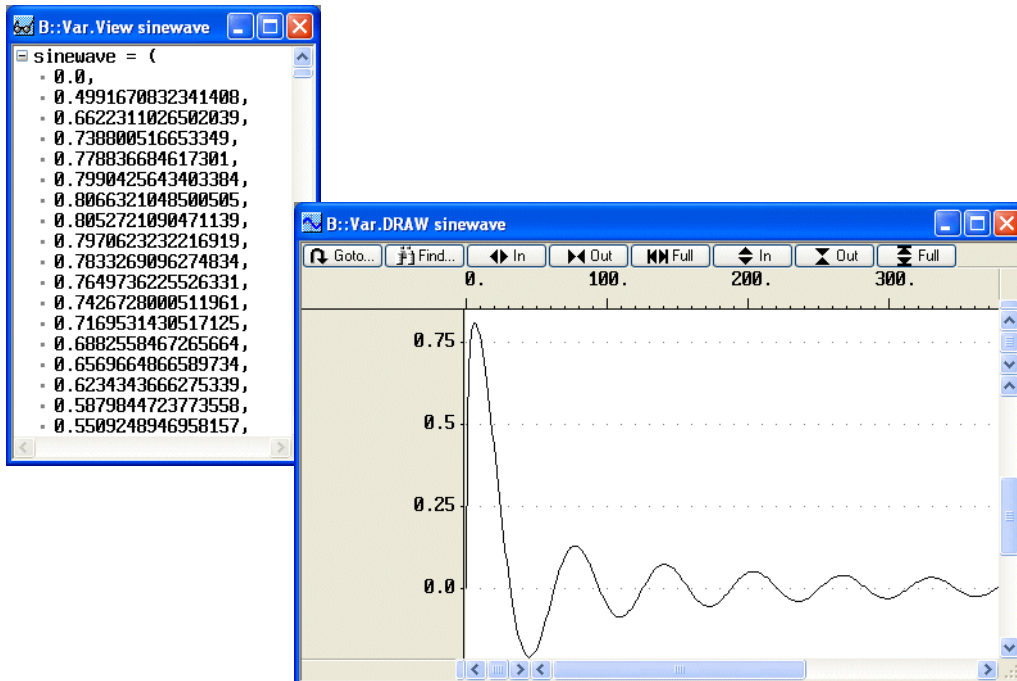
[Args]	引数を表示します。
[Local]	ローカル変数を表示します。
[Caller]	関数のコール元の高級言語ブロックを表示します。

**Var.Frame** [<% フォーマット>] [/ オプション] 'スタックトレース' を表示します。

## グラフ表示

**Var.DRAW** [<% フォーマット>] <配列> 配列の内容をグラフで表示します。

```
Var.DRAW sinewave
```



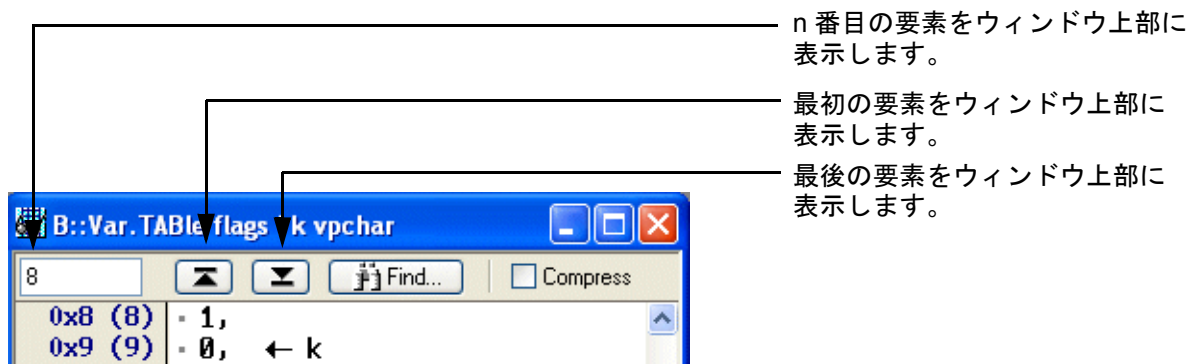
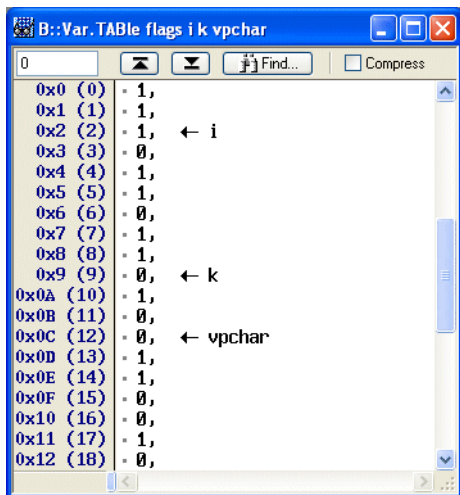
## 配列をインデックスとポインタとともに表示

**Var.TABLE** [<% フォーマット >] <配列 > <インデックス > [...]

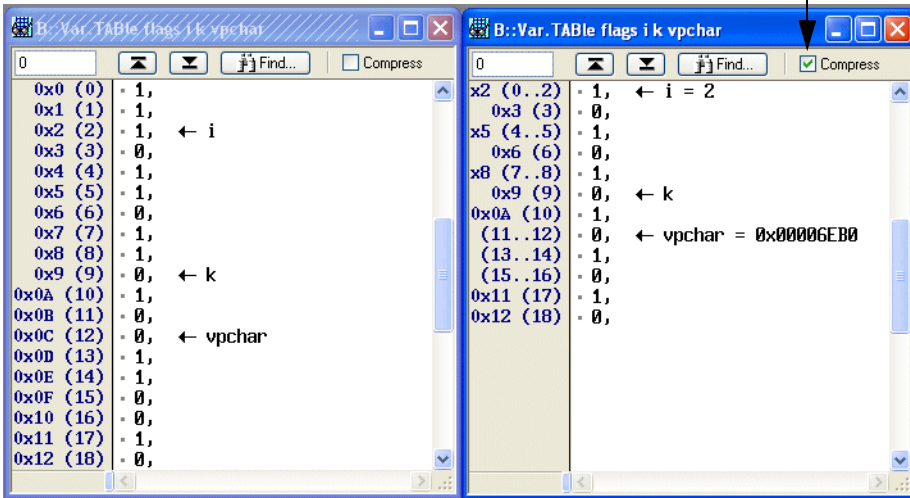
配列をインデックスとポインタとともに表示します。

```
Var.TABLE flags i k vpchar
```

i and k are indices,  
vpchar is a pointer

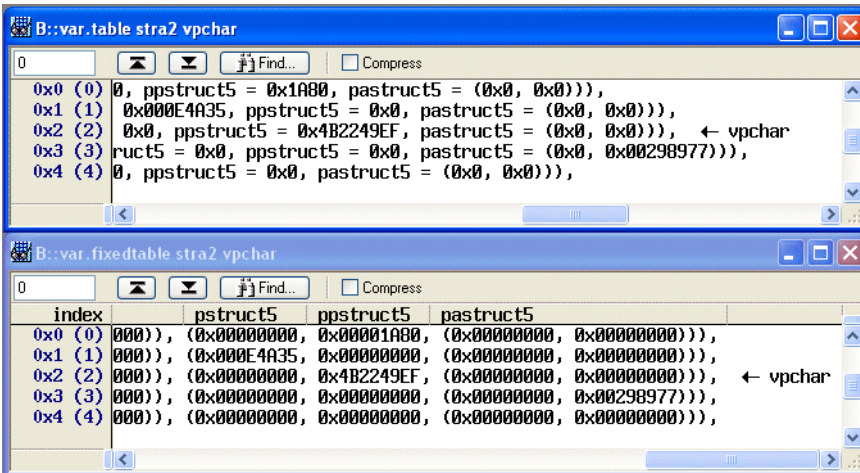


配列表示を圧縮します。



**Var.FixedTable** [<% フォーマット>] <配列> <インデックス> <配列をインデックスとポイントとともに固定フォーマットで表示します。> [...]

Var.FixedTable stra2 vpchar

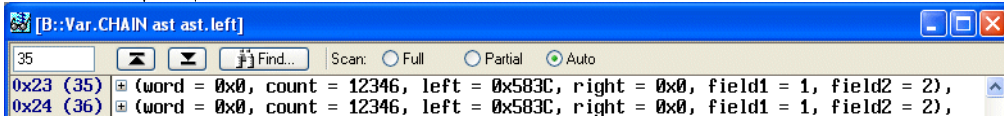
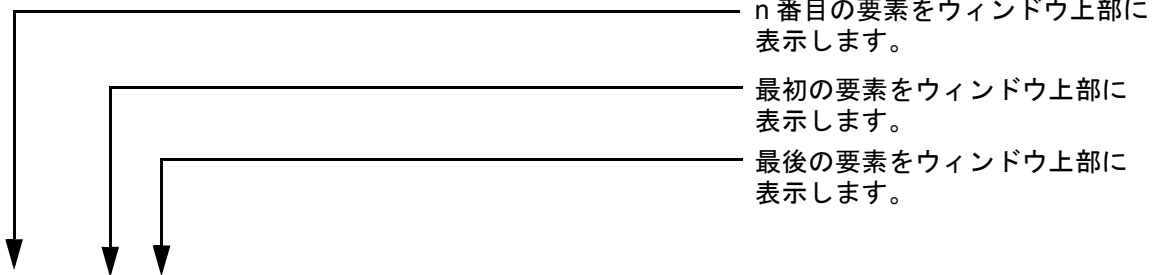
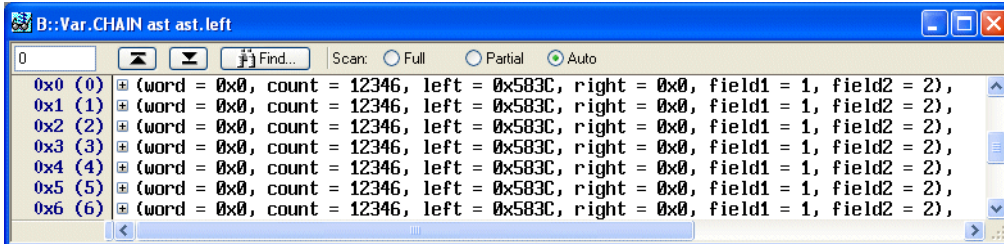


## リンクしたリスト

**Var.CHAIN** [<% フォーマット>] <第 1> <第 2>   リンクしたリストを表示します。  
[ ... ]

Var.CHAIN ast ast.left

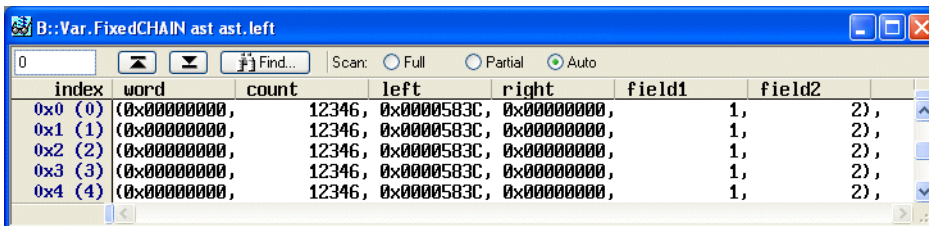
ast is the first element of the  
linked list,  
ast.left provides the pointer to  
the next element



スキャンモード	リンクしたリストを永続的にスキャンして常に最新に保ちます。これにより、TRACE32 ユーザーインターフェースのパフォーマンスが低下する場合があります。3 種類のスキャンモードがサポートされます。
[Full]	リンクしたリストが完全にスキャンされます。これにより、TRACE32 ユーザーインターフェースのパフォーマンスがかなり低下する場合があります。
[Partial]	リンクしたリストが、画面最上部のレコードからのみスキャンされます。TRACE32 ユーザーインターフェースのパフォーマンスへの影響はわずかです。
[Auto]	このモードは、TRACE32 ユーザーインターフェースの速度を犠牲にせずに最新のリンクしたリストを入手する方法です。特定の時間 (20 ~ 50ms) リストが更新され、同じ時間だけユーザーの入力も受け入れます。[Auto] ボタンの横の数字は最後に更新されたレコードの番号です。

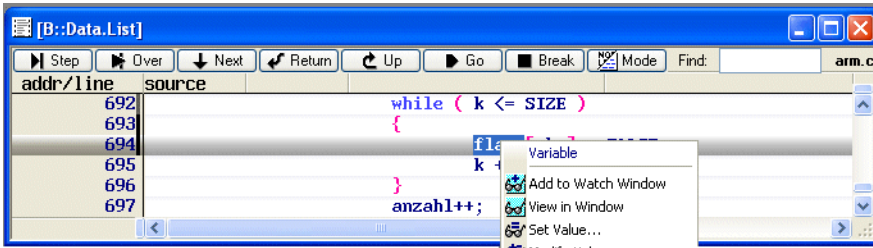
**Var.FixedCHAIN** [<% フォーマット>] <第 1> <第 2> [...]      リンクしたリストを固定フォーマットで表示します。

```
Var.FixedCHAIN ast ast.left
```



# 変数ログ

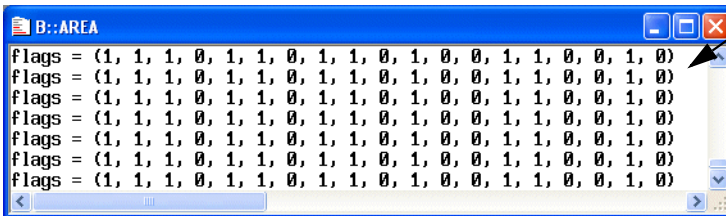
変数ログコマンドは、プログラムが停止するたびに、選択した変数の内容をメッセージエリアに記録します。



ログに記録する変数を選択します。



メッセージエリアを開きます。



プログラムが停止するたびに選択した変数の現在の値がメッセージエリアに記録されます。

**Var.LOG** [<% フォーマット>] [<変数>] ... [/<オプション>] 変数をログに記録します。

**Area.View A000** メッセージエリアを表示します。

- ログをオフにするには、Var.Log をパラメータなしで使用します。

**AREA.Create** < 領域 >

専用のウィンドウを作成します。

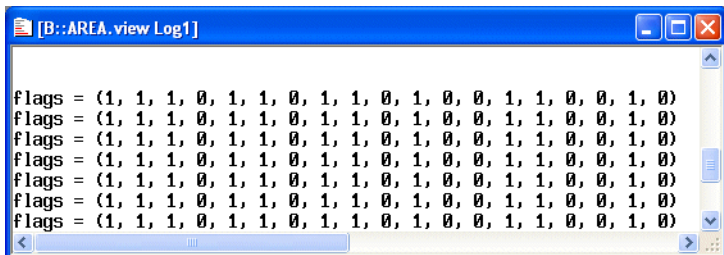
**AREA.view** < 領域 >

専用のウィンドウを表示します。

```
AREA.Create Log1
```

```
AREA.view Log1
```

```
Var.LOG flags /AREA Log1
```



```
AREA.Create Log1
```

```
AREA.view Log1
```

```
Var.LOG flags /AREA Log1 /Changes
```

Variable is only recorded when it has changed since the last program stop

**AREA.OPEN** < 領域 > < ファイル名 >

AREA ウィンドウの出力ファイルを開きます。

**AREA.CLOSE** < 領域 >

AREA ウィンドウの出力ファイルを閉じます。

```
AREA.Create Log1
```

```
Area.view Log1
```

```
AREA.OPEN Log1 loglist.lst
```

```
Var.LOG flags /AREA Log1
```

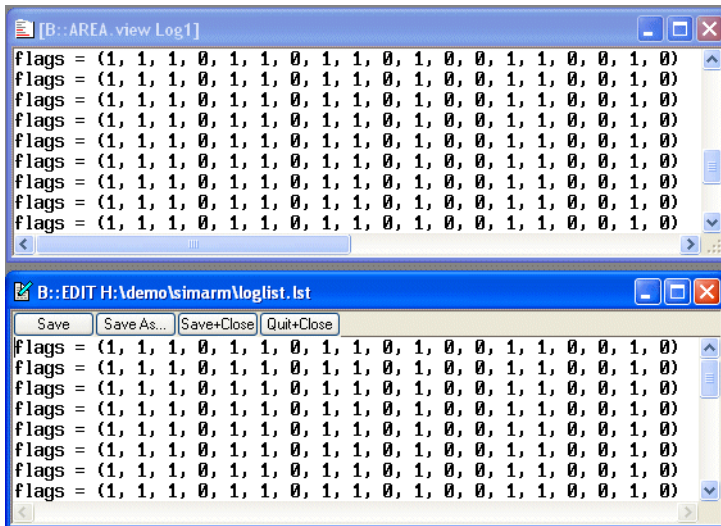
```
AREA.CLOSE Log1
```

```
Var.LOG
```

Finish log

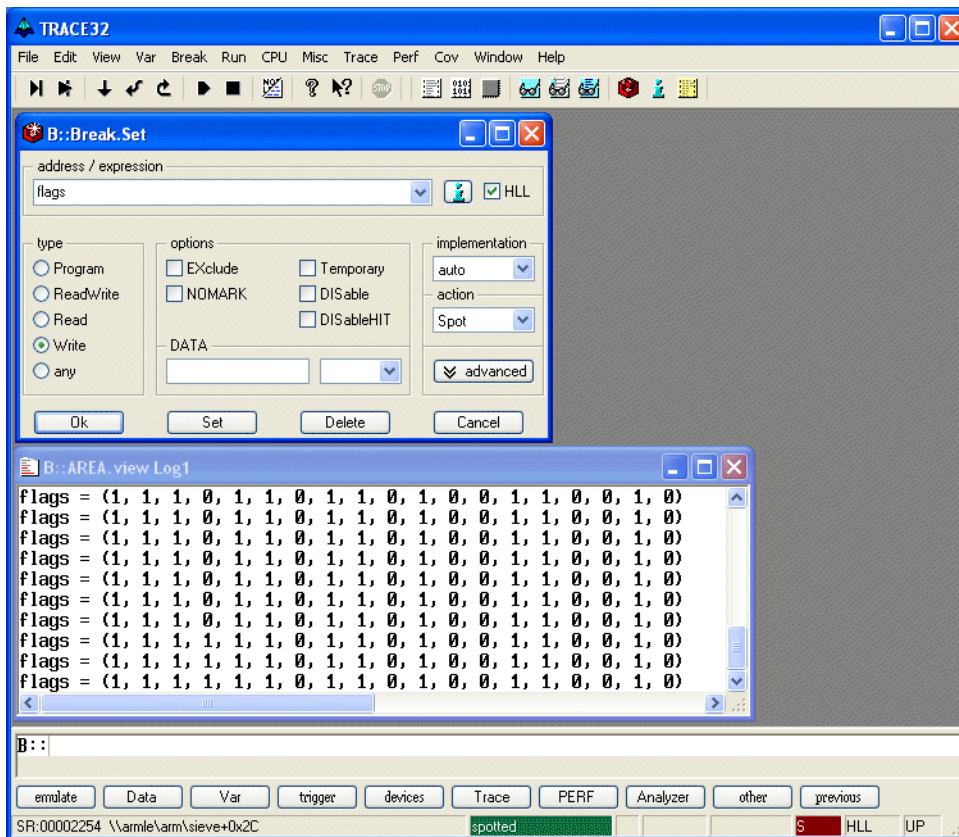
```
EDIT loglist.lst
```

Display the result



## 各スポットポイントの変数のログ

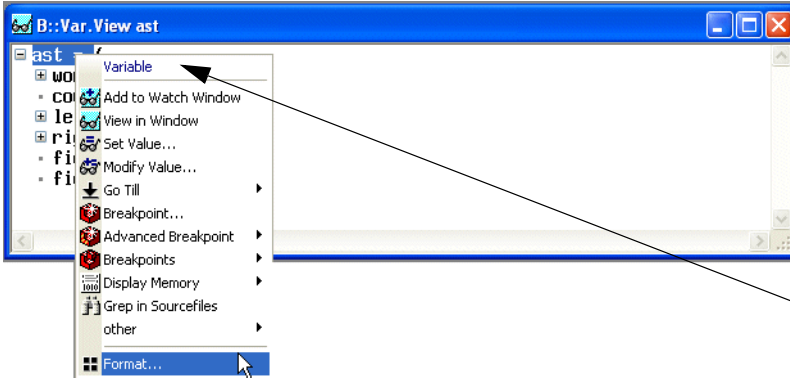
```
Var.Break.Set flags /Write /Spot  
  
AREA.Create Log1  
  
AREA.view Log1  
  
Var.LOG flags /AREA Log1 /ONSPOT
```



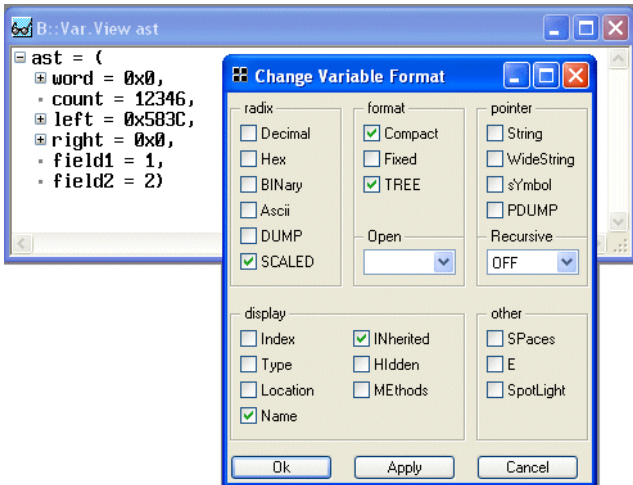
## ログをオフに切り替え

```
Var.LOG Switch log to off
```

## [Format] ダイアログボックスを使用した変数のフォーマット



変数を選択してマウスの右ボタンを押して、**[Change Variable Format]** ダイアログボックスを開きます。

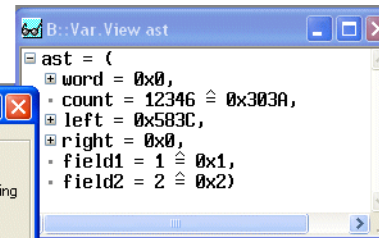
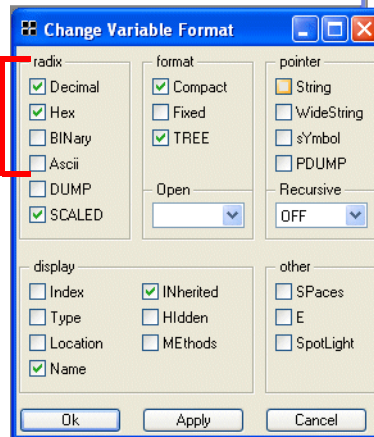


- 数値フォーマット

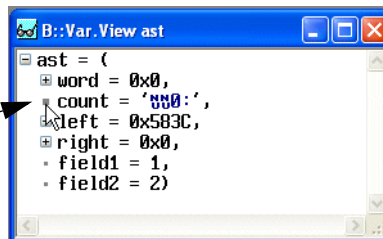
デフォルトでは、整数は 10 進法、ポインタは 16 進法で表示されます。

radix	
[Decimal]	すべての数値が 10 進法で表示されます。
[Hex]	すべての数値が 16 進法で表示されます。
[BINary]	すべての数値が 2 進法で表示されます。
[Ascii]	すべての数値が ASCII 文字で表示されます。

複数の radix を選択できます。

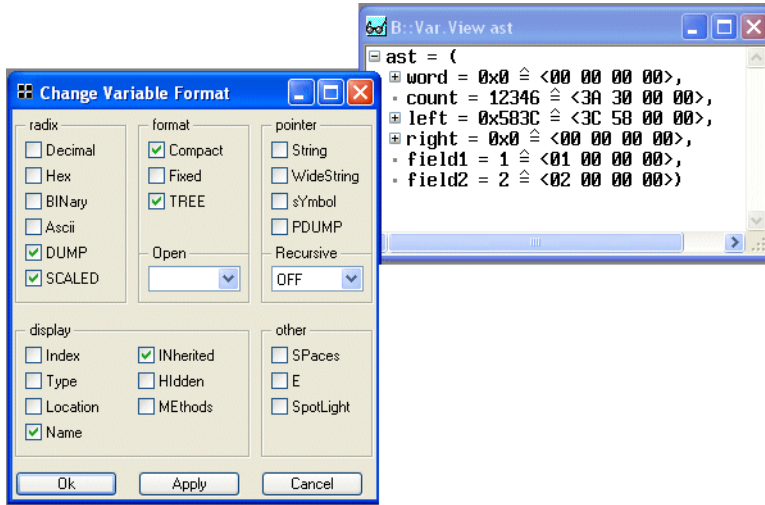


変数の左側の小さな点をクリックして、数値を別のフォーマットで表示します。



- **[DUMP]**

変数の内容をさらに 16 進法ダンプとして表示します。



- **[SCALED]**

変数を定義したスケーリングで表示します。

**sYmbol.AddInfo.Var** < 変数 > **Scaled** < 乗数 > < オフセット > < フォーマット >

変数のスケーリングを定義します。

**sYmbol.AddInfo.List**

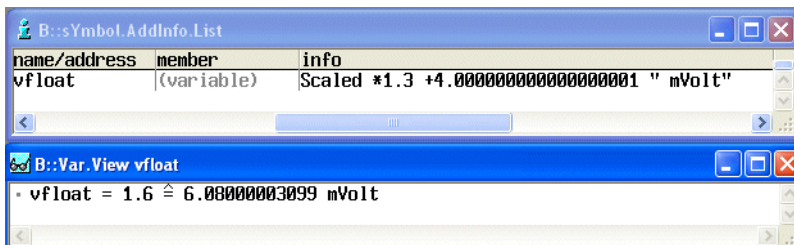
定義したすべてのスケーリングをリストします。

**sYmbol.AddInfo.RESet**

リストをリセットします。

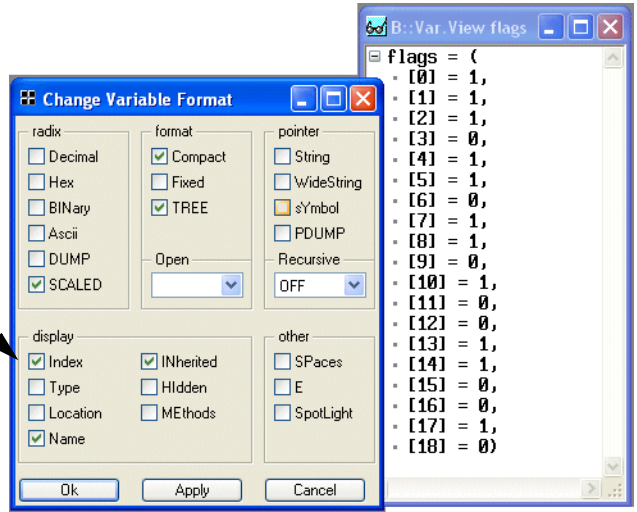
```
sYmbol.AddInfo.Var vfloat Scaled 1.3 4 " mVolt"
```

```
sYmbol.AddInfo.List
```



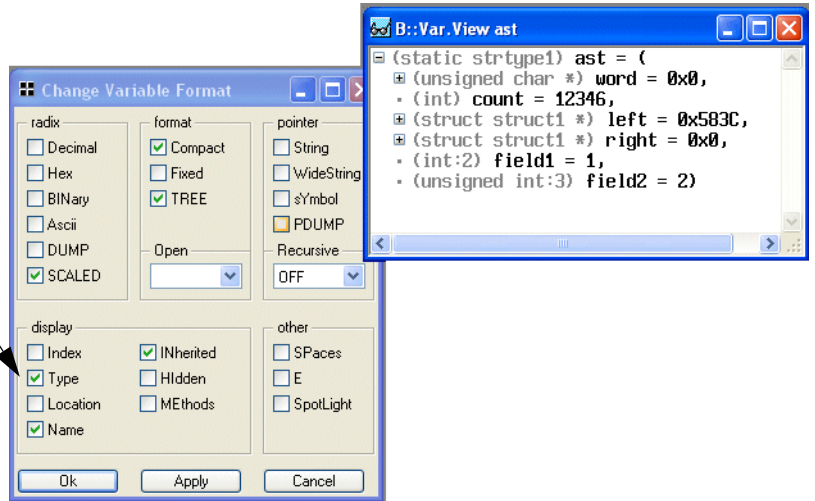
- [Index]

インデックスとともに配列を表示します。



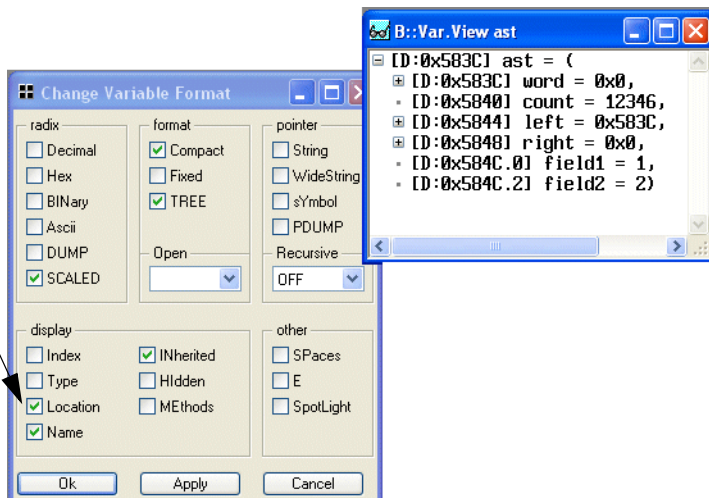
- [Type]

型情報とともに変数を表示します。



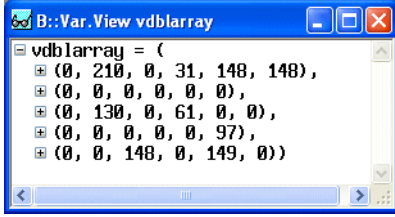
- [Location]

位置の情報とともに  
変数を表示します。

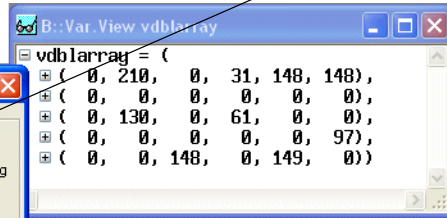
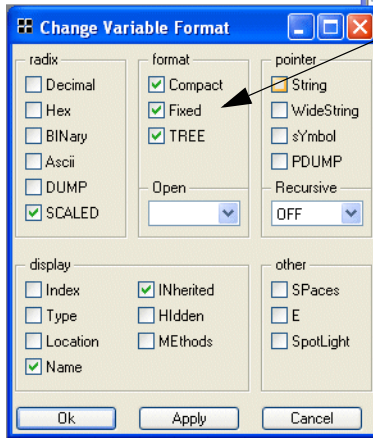


- [Fixed]

すべての数値を固定フォーマットで表示します。

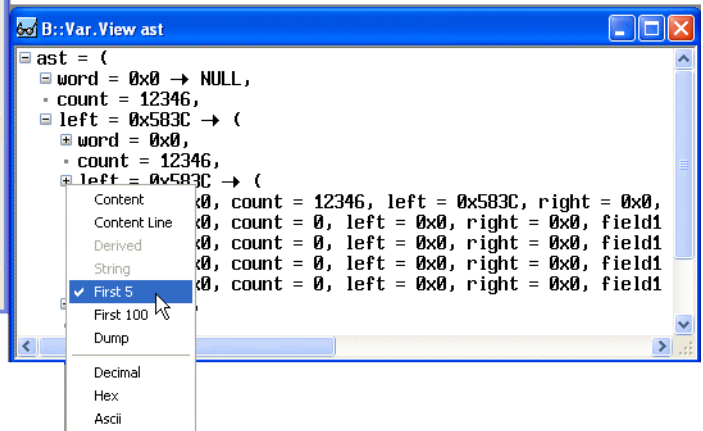
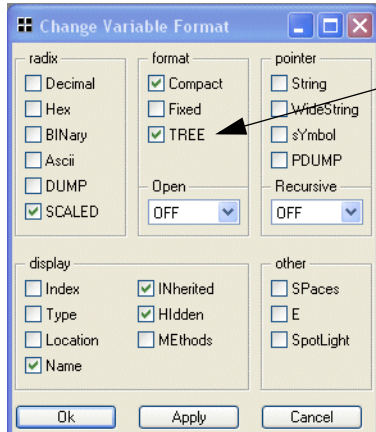


配列の数値要素間に固定スペースを使用します。



- [TREE]

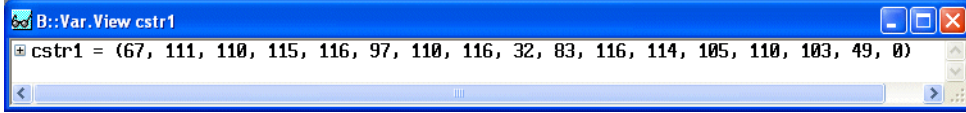
ツリー構造で表示します。



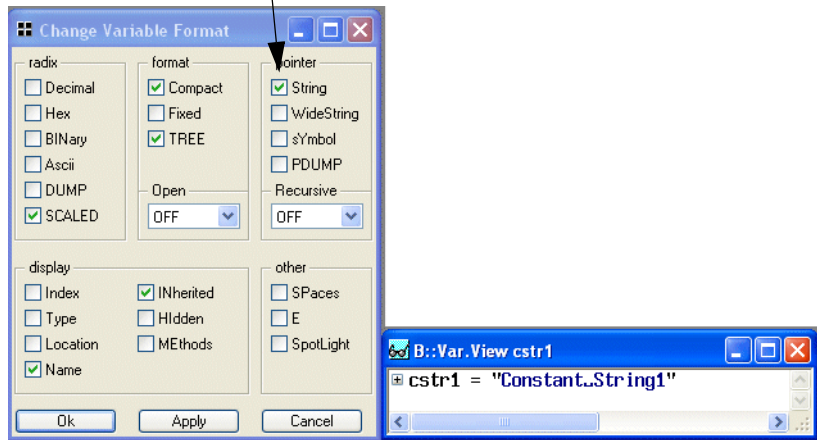
- [String] / [WideString]

このフォーマットは配列または文字へのポインタに使用できます。

[String]	各文字が 1 バイトです。
[WideString]	各文字が 1 ワードです（一部の DSP またはユニコード）。



配列を文字列として表示します。

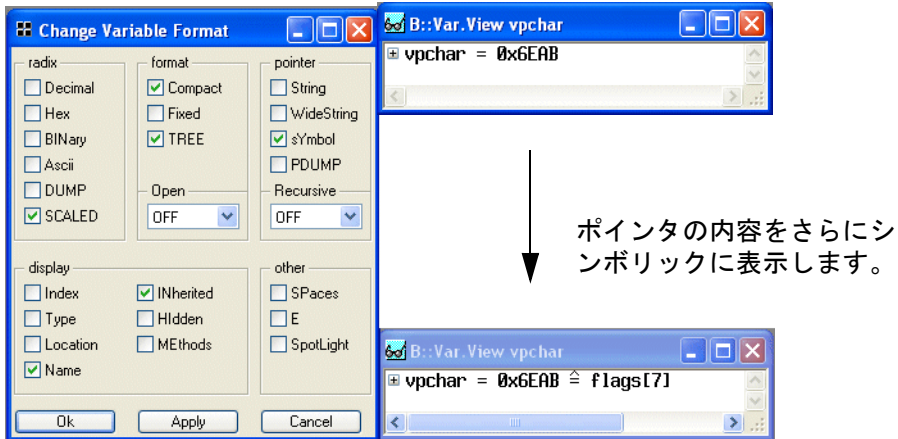


<b>sYmbol.AddInfo.Var &lt; 変数 &gt; ZSTRING</b>	変数の内容を 0 で終わる文字列として定義します。
<b>sYmbol.AddInfo.List</b>	すべての定義をリストします。
<b>sYmbol.AddInfo.RESet</b>	リストをリセットします。

<code>sYmbol.AddInfo.Var cstr1 ZSTRING</code>	The contents of cstr1 is a zero-terminated string
<code>sYmbol.AddInfo.List</code>	Display definition list

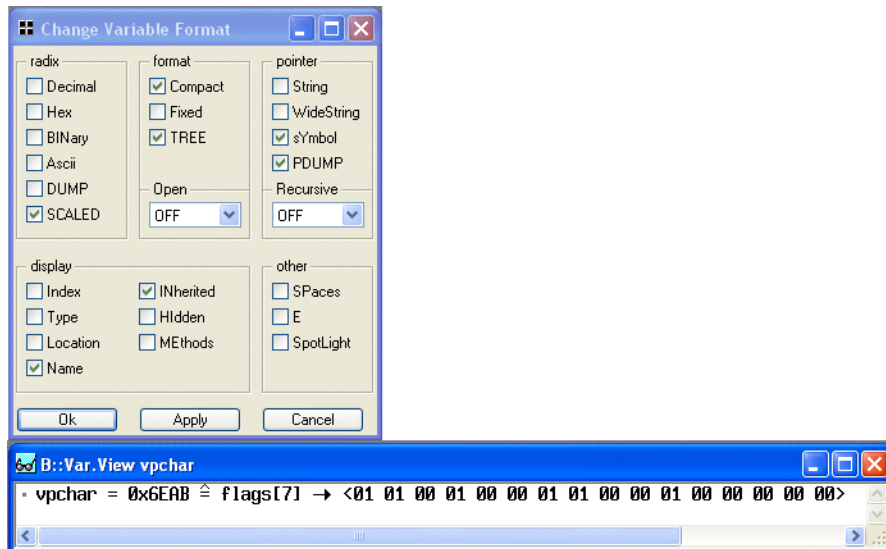


- **[sYmbol]**

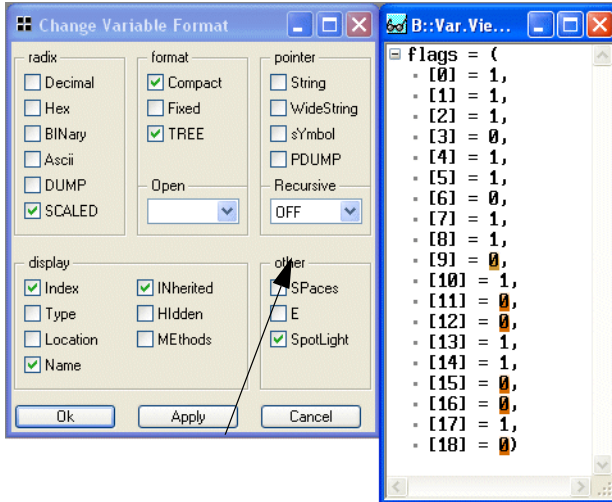


- **[PDUMP]**

ポインタが指すアドレスから開始する 16 バイトの 16 進ダンプを表示します。



## • [SpotLight]

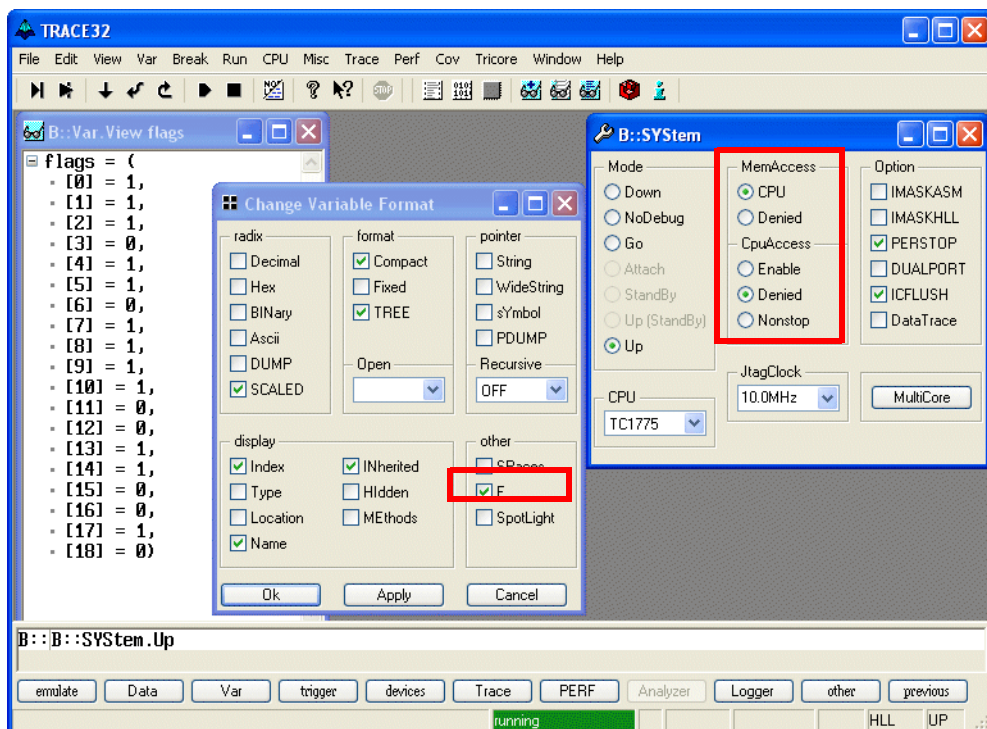


変更された変数要素をすべて強調表示します：

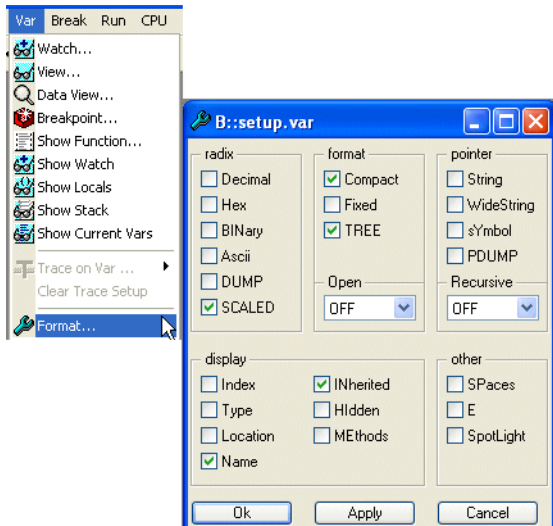
最後のステップで変更された変数要素には濃い赤のマークが付きます。最後のステップの前のステップで変更された変数要素には、それより明るい色のマークが付きます。このように4レベルでカラー表示されます。

- [E] (実行時メモリアクセス)

プログラムの実行中に変数の内容を更新します。これは、[SYSTEM] ウィンドウで SYSTEM.MemAccess CPU/NEXUS または SYSystem.CpuAccess Enable を設定した場合のみ機能します。





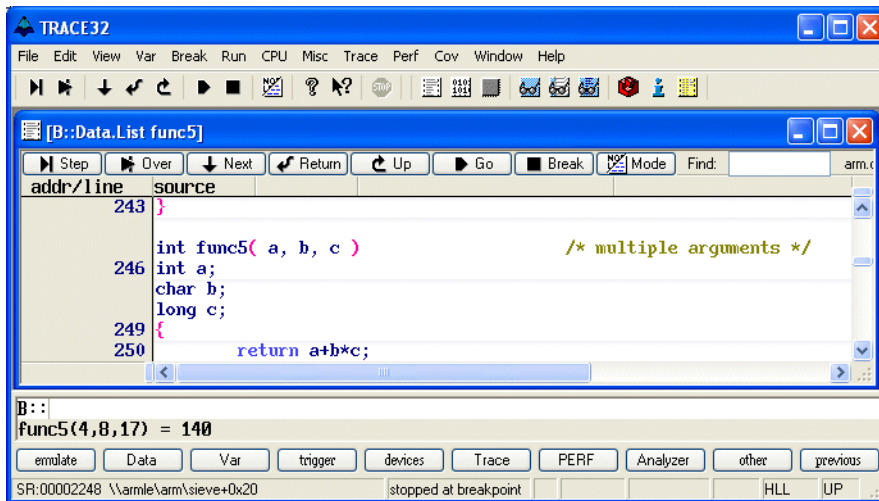


**SETUP.Var** [<% フォーマット>...] 変数のデフォルト表示フォーマットを変更します。

**Var.set** [<% フォーマット>] <変数>

ターゲットで関数を実行します。

```
Var.set func5(4,8,17)
```



**Var.Call** [<% フォーマット>] <式>

1ステップ実行するために、関数とパラメータをコールします。

**Register.SWAP**

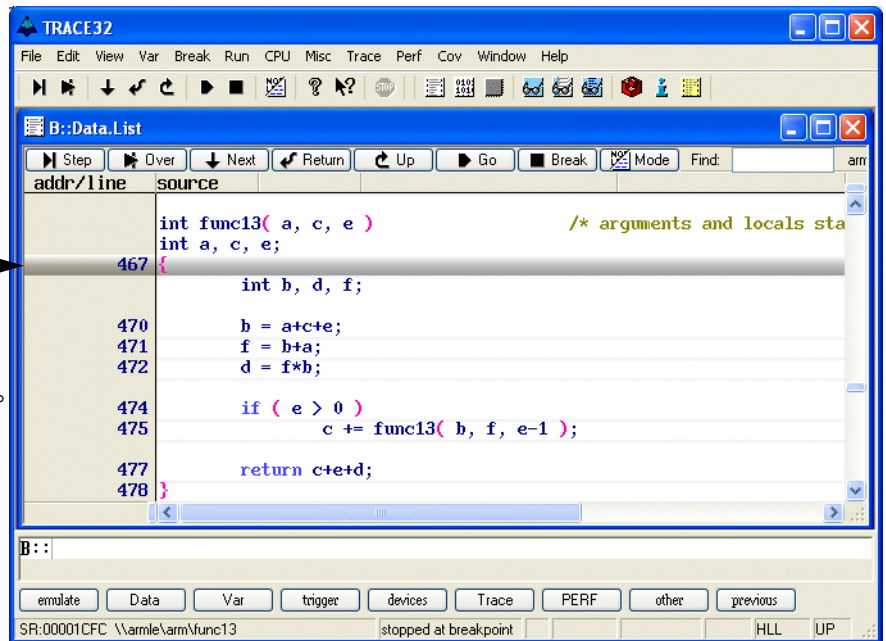
関数の最後で CPU レジスタを再コールします。

式が関数コールの場合は、この関数を入力すると、プログラムカウンタが関数の最初の命令を示します。

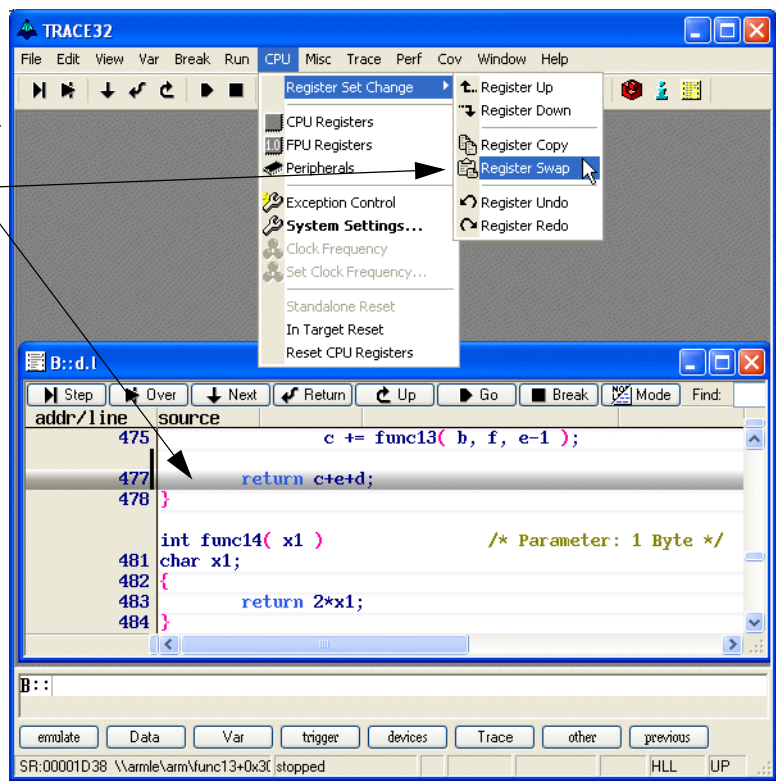
**Register.SWAP** コマンドで、関数コールの前の CPU レジスタの値を再コールすることができます。

```
Var.set func13(3,8,5)
```

プログラムカウンタが関数の最初の命令を示します。これで、関数を1ステップ実行できます。



関数コールのときに情報がスタックに保存されなかったため、[Register Swap]を使用してすべてのCPUレジスタを関数コールの前の値に復元します（returnを実行する前に復元する）。



HLL: トレーニング .....	HLL Debugging - トレーニング	2
トレーニング : HLL .....	HLL Debugging - トレーニング	2