LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide G

Release 02.2023

MANUAL

General Commands Reference Guide G

Version 10-Feb-2023

History

05-Jan-22	Support wildcard in GROUP.CreateTASK.

GLOBALON Global event-controlled PRACTICE script execution

[<events>] [<action>] [Example]

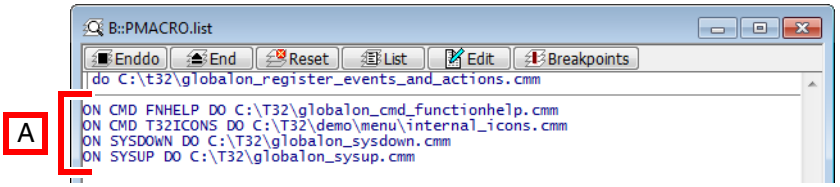
Format:	**GLOBALON** *<event>* [*<action>*]
<event>:	*<device_specific_events>* *<practice_specific_events>* *<cpu_specific_events>*
<device_specific_events>:	**ABREAK** **CORESWITCH** **GO** **PBREAK** **PBREAKAT** *<address>* **POWERDOWN** **POWERUP** **RESET** **SYSDOWN** **SYSUP** **TRIGGER**
<action>:	**DO** *<file>*

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

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

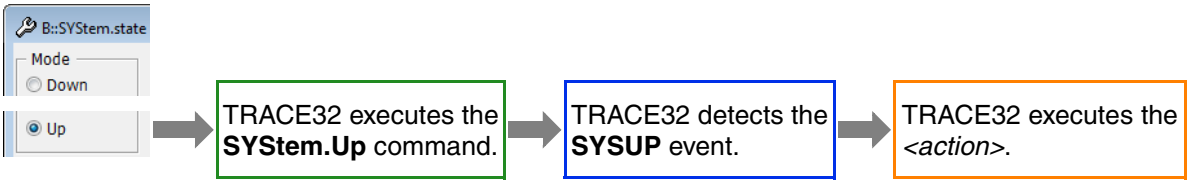
Registered actions remain stored on the global PRACTICE stack frame. Therefore, the actions are valid for the entire duration of the TRACE32 session, or until they are removed manually.

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



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

Let's assume that an action has been registered for the **SYSUP** event. When a **SYStem.Up** command is initiated via the TRACE32 PowerView GUI or the command line or via another PRACTICE script (*.cmm), then TRACE32 responds as illustrated in the figure below:



Events: <device_specific_events>

Device-specific Events	Descriptions
ABREAK	The analyzer mode changed to the break state.
CORESWITCH	SMP debugging: The currently displayed context changed to a different core or thread.
GO	The target program started.
PBREAK	The target program stopped.
PBREAKAT	The target program stopped at a specific address.
POWERDOWN	Target power is switched off.
POWERUP	Target power is switched on.
RESET	A target reset was detected.
SYSDOWN	System mode changed to **Down** or **NoDebug**. The event is also triggered if the debugger is in system mode **StandBy** and the target power is switched off.
SYSUP	System mode changed to **Up**. The event is also triggered if the debugger is in system mode **StandBy** and the target power is switched on.
TRIGGER	A podbus trigger occurred (internal or external source can be selected via TRIGGER window).

Events: <practice_specific_events>

<practice_specific_ events>	For a description of the PRACTICE specific events, such as **GLOBALON ERROR**, refer to **GLOBALON** (practice_ref.pdf).

Events: <cpu_specific_events>

<cpu_specific_ events>	For information about CPU specific events, refer to the **Processor Architecture Manuals** [▲] listed in the **See also** block below.

<actions>

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

Actions	Descriptions
no action specified	An already defined action for a particular global event will be removed from the global PRACTICE stack frame. See “Unregistering GLOBALON Commands”.
DO	If the event occurs, the specified PRACTICE script file will be executed automatically.

1. Develop the action (PRACTICE script *.cmm) you want to be executed automatically whenever the desired event occurs.

   For demo purposes, we will use two simple scripts for the events SYSUP and SYSDOWN so that you can reproduce the example right away.

   globalon_sysup.cmm

   ```
   PRINT "System up at " Clock.Time()
   AREA ; Display the message in the AREA window

   ; Other commands such as Data.Set, PER.Set to disable an
   ; external watchdog
   ; ...

   ENDDO
   ```

   globalon_sysdown.cmm

   ```
   PRINT "System down at " Clock.Time()
   AREA ; Display the message in the AREA window
   ; ...
   ENDDO
   ```

2. Register the events and their actions in TRACE32.

   ```
   ; At the global PRACTICE stack frame, the following
   ; device-specific events are registered: SYSUP and SYSDOWN

   ; On SYSUP, this PRACTICE script file (*.cmm) is called:
   GLOBALON SYSUP DO "~~/globalon_sysup.cmm"

   ; On SYSDOWN, this PRACTICE script file (*.cmm) is called:
   GLOBALON SYSDOWN DO "~~/globalon_sysdown.cmm
   ```

The path prefix `~~/` works on Windows and Linux and expands to the system directory of TRACE32, by default `C:/T32` for Windows.

Unregistering GLOBALON Commands

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

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

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

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

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

```
END ; Ends all active PRACTICE scripts

; Unregisters the action for the SYSDOWN event
GLOBALON SYSDOWN    ; Do not include the DO <action> here!
```

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

See also

■ ON ■ END ■ PMACRO.RESet

▲ 'Mico32 specific Event for the ON and GLOBALON Command' in 'Mico32 Debugger'
▲ 'CPU specific Events for the ON and GLOBALON Command' in 'Intel® x86/x64 Debugger'

Go

Debug control, program execution, and real-time emulation

See also

- Go.Asm
- Go.Change
- Go.Mix
- Go.Till
- List
- Go.Back
- Go.direct
- Go.MONitor
- Go.TillWarning
- Step
- Go.BackEntry
- Go.Hll
- Go.Next
- Go.Up
- Go.BackTillWarning
- Go.Java
- Go.Return
- Break

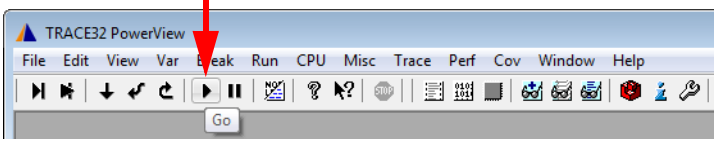
▲ 'Release Information' in 'Legacy Release History'

Debug Control for Debuggers

The command **Go** starts the program execution on the chip/core. By default the program is executed in real-time, but there are features within TRACE32 that suspend the real-time execution. Examples are:

- **Intrusive breakpoint**
- **Performance analysis** via **StopAndGo**

Go command in the toolbar



NOTE:	**Go** is *not* equivalent to the **SYStem.Mode Go** command. **SYStem.Mode Go** resets the processor/chip, enables the on-chip debug logic, and then starts the program execution.

Restarting from Breakpoint

When interrupts are pending and the program execution is started from a breakpoint, it is possible that the processor/core executes the interrupt service routine and returns to the same breakpoint location afterward. The debugging seems to stick on the breakpoints.

To avoid this behavior, TRACE32 executes a single step when the program execution is started on a breakpoint if required. However, this strategy does not solve the problem completely. To completely solve the issue, you have to disable the interrupts will single stepping on assembler level with the TRACE32 command **SYStem.Option.IMASKASM ON**.

SYStem.Option.IMASKASM ON is not a default setting, because it may disturb debugging parts of the program (e.g. a boot loader) that enable/disable interrupts.

Format:	**Go.Asm** [*<address>* [/*<breaktype>* …]] …
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write**
	Onchip | **HARD** | **SOFT**
	ProgramPass | **ProgramFail**
	MemoryReadWrite | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* …
	Alpha | **Beta** | **Charly** | **Delta** | **Echo**
	WATCH | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**
	Spot **DISable** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME**
	SingleCORE (SMP debugging only)

Starts the program execution and switches the **debug mode** to Asm mode.

<breaktype> For a description of the breakpoint types and breakpoint options, see **Break.Set**.

If one or more addresses are specified, temporary breakpoints are set before the program execution is started.

```
Go.Asm                          ; switch to debug mode assembler and
                                ; start the program execution
Break                           ; stop the program execution


Go.Asm d_add                    ; set a temporary Program breakpoint to
                                ; the label d_add, switch to debug mode
                                ; assembler and start the program
                                ; execution

Go.Asm D:0x40004128 /Write      ; set a temporary Write breakpoint to
                                ; the address D:0x40004128, switch to
                                ; debug mode assembler and start the
                                ; program execution
```

See also

- Go
- Go.direct

Format: **Go.Back** [*<address>* [/*<breaktype>* …]] …

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

Re-runs the recorded trace information backward until the specified point (only for trace-based debugging - **CTS**).

<breaktype> For a description of the breakpoint types and breakpoint options, see **Break.Set**.

Example:

```
Trace.List                      ; open a Trace Listing

CTS.GOTO -22918643.             ; specify record -22918643. as CTS
                                ; starting point

Go.Back func13                  ; re-run the recorded trace information
                                ; backward until the entry to func13
```

See also

■ Go ■ Go.direct ■ CTS

▲ 'Release Information' in 'Legacy Release History'

Go.BackEntry Go back in program to function entry (CTS)

Format:	**Go.BackEntry /Endless**

Re-runs the recorded trace information backward until the entry of the current function (only for trace-based debugging - **CTS**).

Example:

```
Trace.List                      ; open a Trace Listing

CTS.GOTO -22918643.             ; specify record -22918643. as CTS
                                ; starting point

Go.BackEntry                    ; re-run the recorded trace information
                                ; backward until the entry of the current
                                ; function
```

See also

■ Go ■ Go.direct

▲ 'Release Information' in 'Legacy Release History'

Go.BackTillWarning — Go back in program until warning (CTS)

Format:	**Go.BackTillWarning**

Re-runs the recorded trace information backward until the previous warning (only for trace-based debugging - **CTS**). An explanation for the warning is given in the message area. A full example is given at **Go.TillWarning**.

See also

- Go
- Go.direct
- CTS.state

Go.Change — Run program till content changes

Format:	**Go.Change** *<content>*

Starts the program execution. Whenever a breakpoint is hit, check if *<content>* changed. If *<content>* has not changed, re-start program execution automatically.

Example:

```
Break.Set 0x100              ; set a Program breakpoint at address 0x100
Break.Set 0x200              ; set a Program breakpoint at address 0x200
Go.Change Register(R31)      ; starts the program execution
                             ; check at each breakpoint hit if the
                             ; content of register R31 changed
                             ; if not, re-start the program execution
                             ; automatically
```

See also

- Go
- Go.direct

[Examples]

Format:	**Go.direct** [*<address>* [/*<breaktype>* …]] …
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **MemoryReadWrite** | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* … **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **Spot** **DISable** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME** **SingleCORE** (SMP debugging only)

Starts the program execution. If one or more addresses are specified temporary breakpoints are set, before the program execution is started.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.
SingleCORE	SMP debugging only: Start program execution only on the currently selected core.

Examples:

```
Go                      ; start program execution

Go func0 func12         ; set temporary breakpoints to the entry of
                        ; function func0 and func12 and then start the
                        ; program execution
                        ; temporary breakpoints are only valid until the
                        ; program execution stops the next time
```

```
CORE.select 1.          ; select core 1

Go /SingleCORE          ; start program execution on
                        ; core 1. only
```

The **Cores** field of the TRACE32 state line displays the number of the currently selected core.

See also

- Go
- Go.Asm
- Go.Back
- Go.BackEntry
- Go.BackTillWarning
- Go.Change
- Go.Hll
- Go.Java
- Go.Mix
- Go.MONitor
- Go.Next
- Go.Return
- Go.Till
- Go.TillWarning
- Go.Up
- Break.direct
- Register.view
- Register()
- STATE.RUN()

▲ 'Release Information' in 'Legacy Release History'

Format:	**Go.HII** [*<address>* [/*<breaktype>* …]] …
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write**
	Onchip | **HARD** | **SOFT**
	ProgramPass | **ProgramFail**
	MemoryReadWrite | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* …
	Alpha | **Beta** | **Charly** | **Delta** | **Echo**
	WATCH | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**
	Spot **DISable** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME**
	SingleCORE (SMP debugging only)

Starts the program execution and switches the **debug mode** to HLL mode. If one or more addresses are specified, temporary breakpoints are set before the program execution is started.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

See also

- Go
- Go.direct

Format:	**Go.Java**

Starts the program execution and stops at the first JAVA byte code to be executed. This command can be used to switch from native debugging to JAVA byte code debugging.

See also

- Go
- Go.direct

Format: **Go.Mix** [*<address>* [/*<breaktype>* …]] …

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

SingleCORE (SMP debugging only)

Starts the program execution and switches the **debug mode** to Mix mode. If one or more addresses are specified temporary breakpoints are set, before the program execution is started.

<breaktype> For a description of the breakpoint types and breakpoint options, see **Break.Set**.

See also

- Go
- Go.direct

Format:	**Go.MONitor**

Starts the program execution and switches to run mode debugging. In run mode debugging all debug events are handled by a so-called debug monitor.

Please be aware that run-mode debugging has to be configured, before it can be used. Typical commands are:

```
SYStem.PORT 10.1.2.99:2345          ; configure the TCP/IP
                                    ; communication to the debug
                                    ; monitor
Go.MONitor
```

```
SYStem.MemAccess GdbMON             ; use Debug Communication Channel
                                    ; (DCC) to communicate with GDB
Go.MONitor
```

The command **Break.MONitor** can be used to switch back to stop mode debugging if this is possible within your debug environment.

See also

- Go
- Go.direct
- Break.MONitor
- Break.SetMONitor

Go.Next Start program and stop at next line

Format:	**Go.Next**

Start the program execution and set a temporary breakpoint set to the next assembler or HLL line. This command can be used to leave a loop or to overstep a subroutine call instruction (see also the command **Step.Over**.)

See also

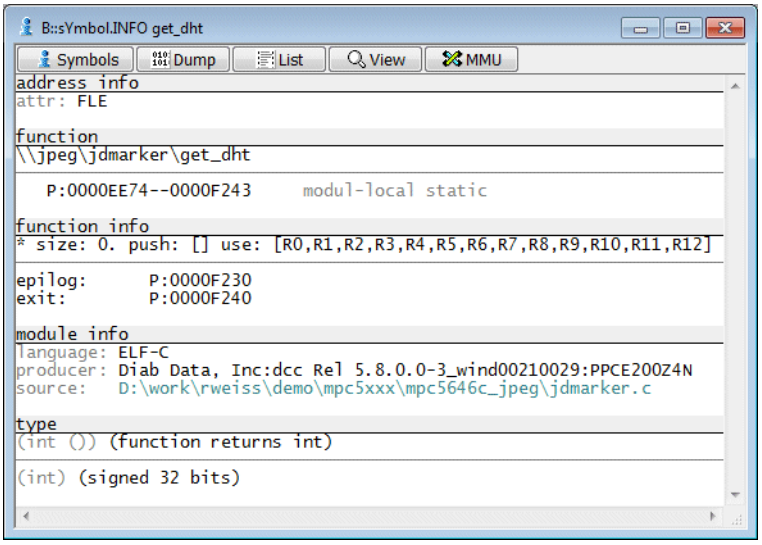
- Go
- Go.direct

Format:	**Go.Return**

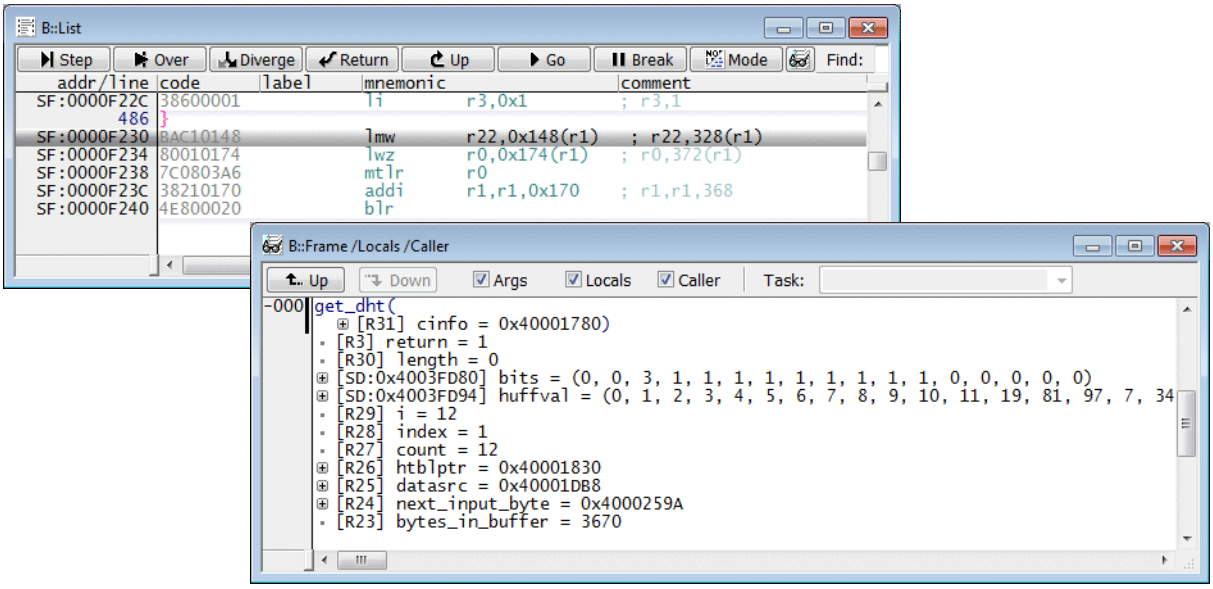
The first **Go.Return** stops at the function epilog, the second **Go.Return** stops at the return of the function. Stopping at the function epilog first has the advantage that the local variables are still valid at this point.

This works in detail as follows:

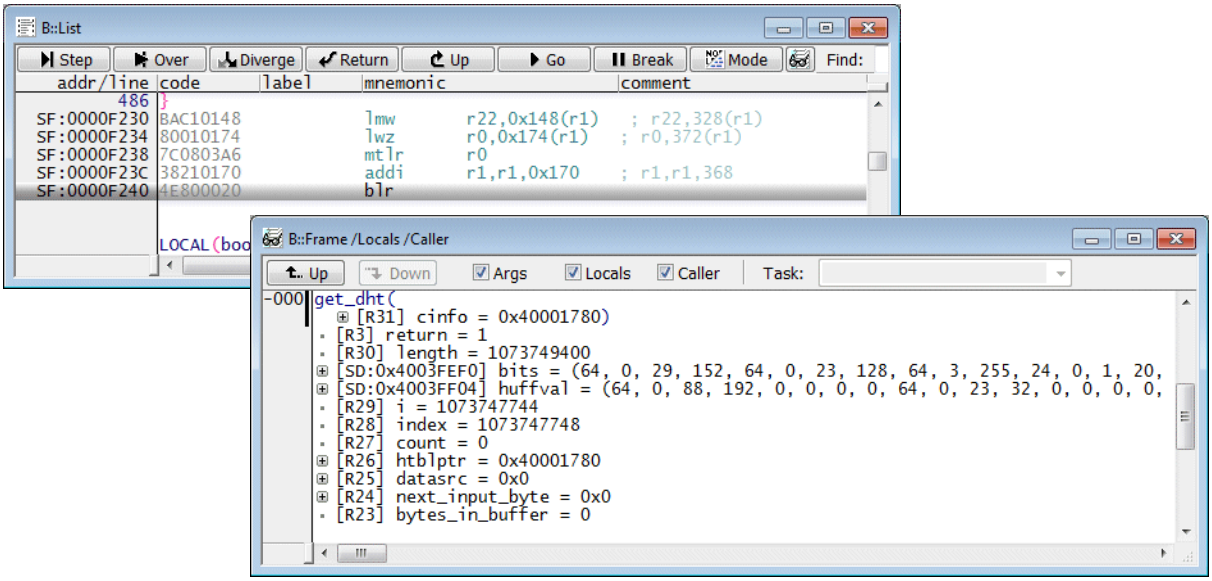
The debug information for a function includes the epilog and exit information (command **sYmbol.INFO**); **epilog** shows the start address of the function epilog, **exit** shows the address of the return of the function.



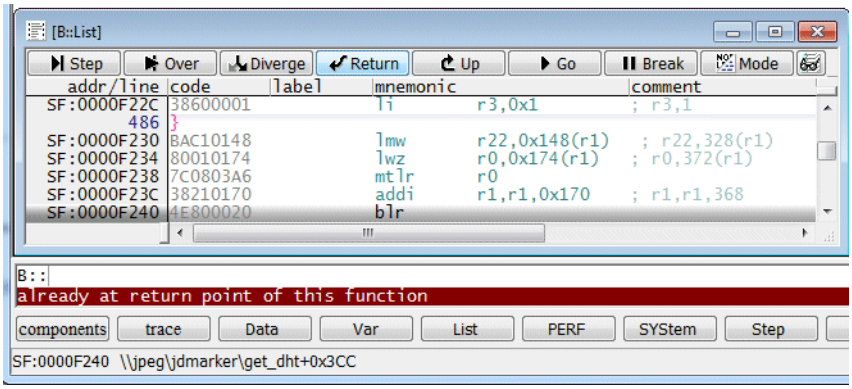
```
Go get_dht                  ; set a temporary breakpoint to the function
…                           ; get_dht and start the program execution
                            ; -> the program execution is stopped at the
                            ; function entry

Step.single                 ; step inside function
Step.single
Go.Return                   ; set a temporary breakpoint to the start address
                            ; of the function epilog and start the program
                            ; execution
                            ; -> the program execution is stopped at the
                            ; function epilog, here all local variables are
                            ; still valid
```




```
Go.Return                   ; set a temporary breakpoint to the return of the
                            ; function and start the program execution
                            ; -> the program execution is stopped at the
                            ; function exit, since the function epilog
                            ; already cleaned the frame pointer, local
                            ; variables are no longer valid
```




```
Go.Return                ; if the command Go.Return is used when the
                         ; the instruction pointer is already at the
                         ; return of the function, an error message is
                         ; generated
```



See also

■ Go ■ Go.direct

▲ 'Release Information' in 'Legacy Release History'

Format:	**Go.Till** *<boolean_expression>*

Starts the program execution. Whenever a breakpoint is hit, **Go.Till** checks if the *<boolean_expression>* became true. If not, **Go.Till** re-starts the program execution automatically.

Example:

```
Break.Set 0x100                        ; set a Program breakpoint at
                                       ; address 0x100
Break.Set 0x200                        ; set a Program breakpoint at
                                       ; address 0x200
Go.Till Data.Byte(D:0x100)==0x0        ; start the program execution,
                                       ; check at each breakpoint hit if
                                       ; the content of the byte at
                                       ; address 0x100 is 0
                                       ; if not, re-start the program
                                       ; execution automatically
```

See also

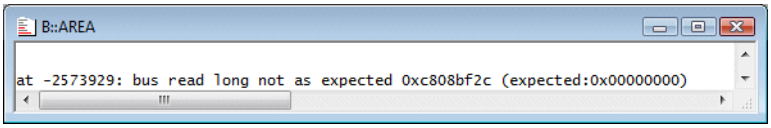
- Go
- Go.direct

Format:	**Go.TillWarning**

Re-runs the recorded program flow until the next warning (only for trace-based debugging - **CTS**).

An example for a warning is given in the message area.

```
AREA.view                          ; open message area

Trace.List                         ; open a Trace Listing

CTS.GOTO -17281536.                ; specify record -17281536. as CTS
                                   ; starting point

CTS.state                          ; open the CTS state window and
                                   ; and check for warnings

Go.TillWarning                     ; re-run the recorded program
                                   ; until the next warning
```



See also

- Go
- Go.direct
- CTS.state

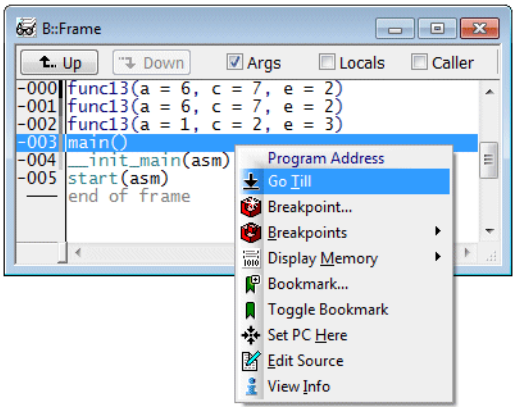
Format:	**Go.Up** [*<level>* | *<address>*]

Starts the program execution in order to return to the caller function. A temporary breakpoints is set directly behind the function call in the caller function.

Without arguments it returns to the function that called the current function (level 1).

<level>

With a *<level>* argument it starts the program execution in order to return 3 levels up in the call hierarchy (see also command **Frame.view**).



<address>

With an *<address>* argument it returns to the first function on the call stack, which includes the given address. The address can be defined symbolically, by the name of the function, or by a line number within the function.

```
Go.Up                ; return to the caller of the current function

Go.Up 3.             ; return three levels up in the function nesting

Go.Up main           ; return to function main
```

See also

- Go
- Go.direct

GROUP

Group functions, modules, or tasks

The **GROUP** command group allows to structure application programs consisting of a huge number of functions/modules/tasks to ease the evaluation of the trace contents and the debugging process.

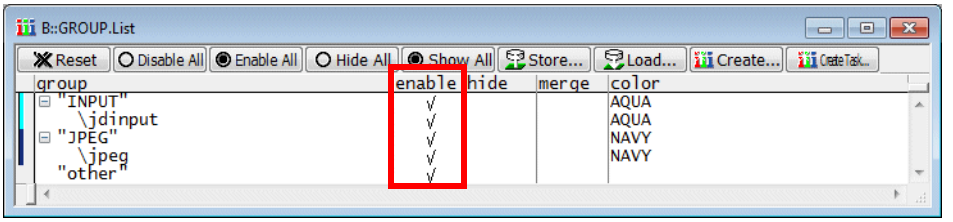
See also

- GROUP.COLOR
- GROUP.Create
- GROUP.CreateFunctions
- GROUP.CreateLabels
- GROUP.CreateModules
- GROUP.CreatePrograms
- GROUP.CreateSources
- GROUP.CreateTASK
- GROUP.Delete
- GROUP.DeleteTASK
- GROUP.DISable
- GROUP.ENable
- GROUP.HIDE
- GROUP.List
- GROUP.Merge
- GROUP.RESet
- GROUP.SEParate
- GROUP.SHOW
- GROUP.EXIST()

▲ 'GROUP Function' in 'General Function Reference'

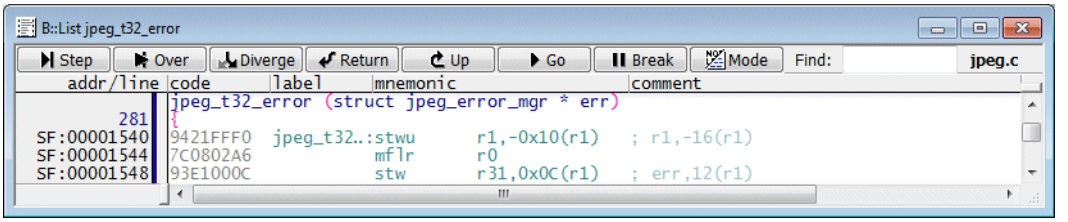
Features

ENable

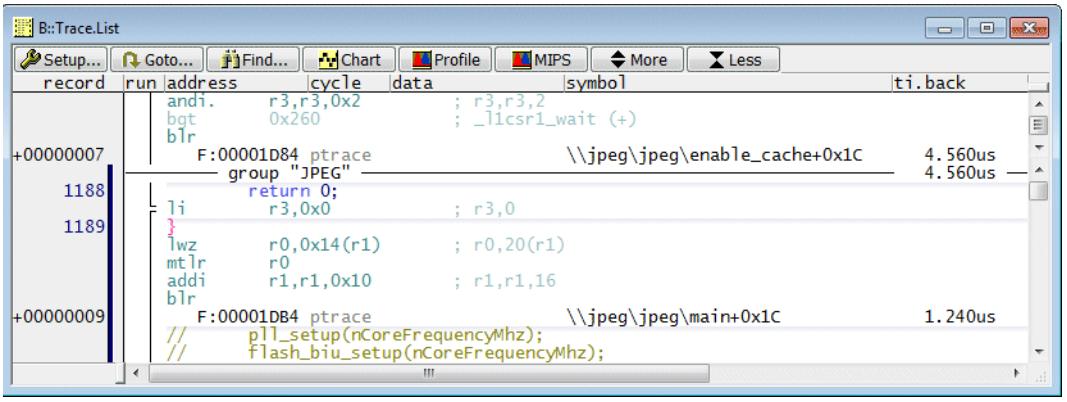


TRACE32 PowerView provides the following features if a group is enabled:

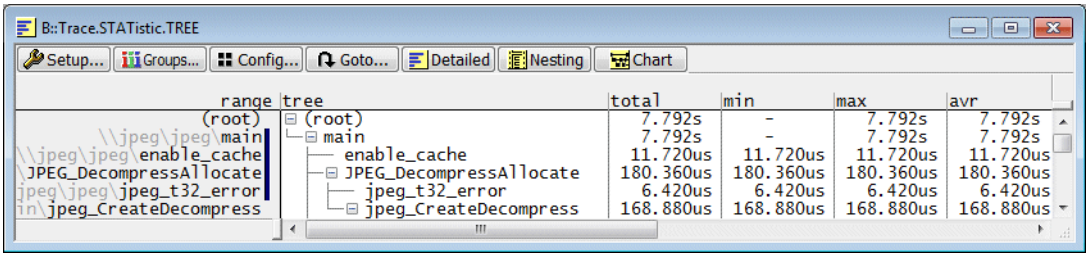
- The source code of all group members is marked with the color assigned to the group.



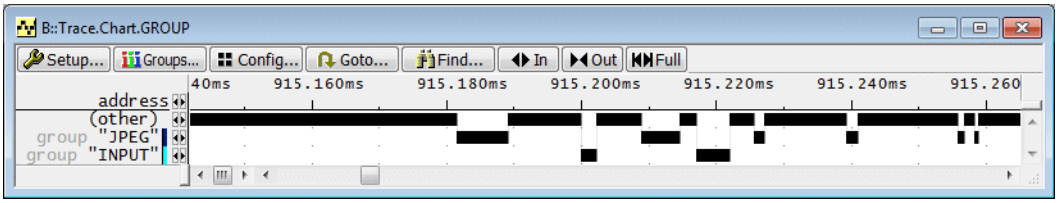
- The trace information recorded for the group members is marked with the color assigned to the group.



- All group members are marked with the color assigned to the group in all trace analysis windows.

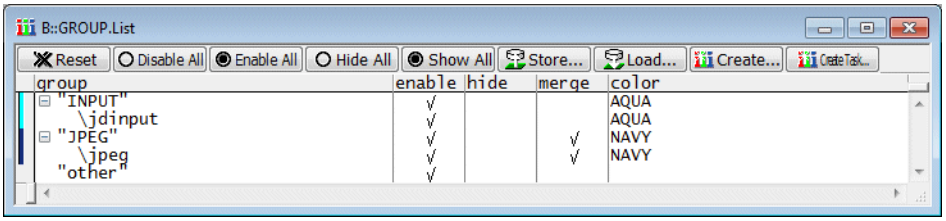


- Additional group-based trace analyses commands are provided.

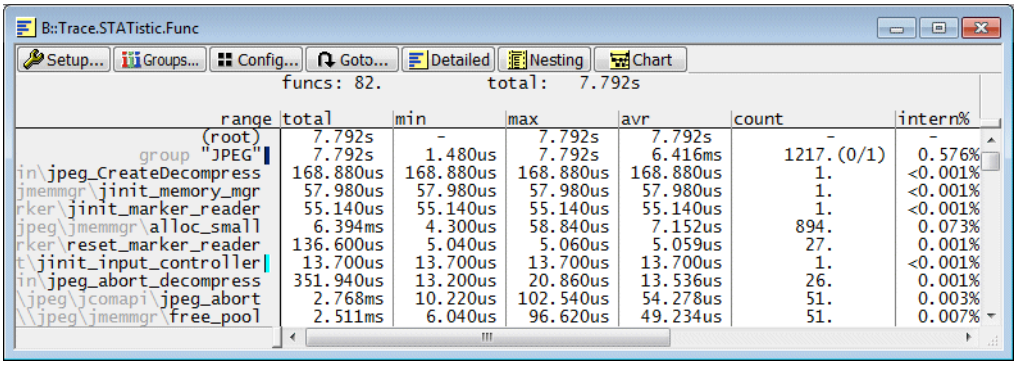


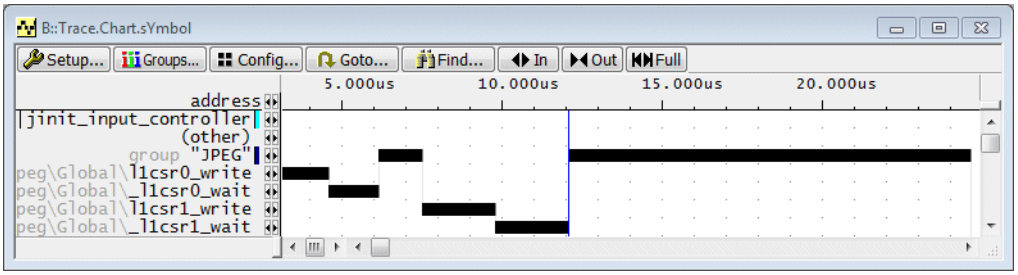
Trace.STATistic.GROUP	Group-based run-time analysis.
Trace.Chart.GROUP	Group time chart.
Trace.PROfileChart.GROUP	Group profile chart.
MIPS.STATistic.GROUP	MIPS statistic for groups.
MIPS.PROfileChart.GROUP	MIPS profile chart for groups.

If a group is enabled, the following features are added by checking merge:

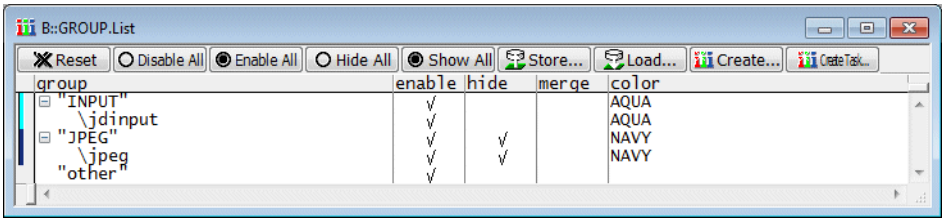


- The group represents its members in all trace analysis windows. No details about group members are displayed.

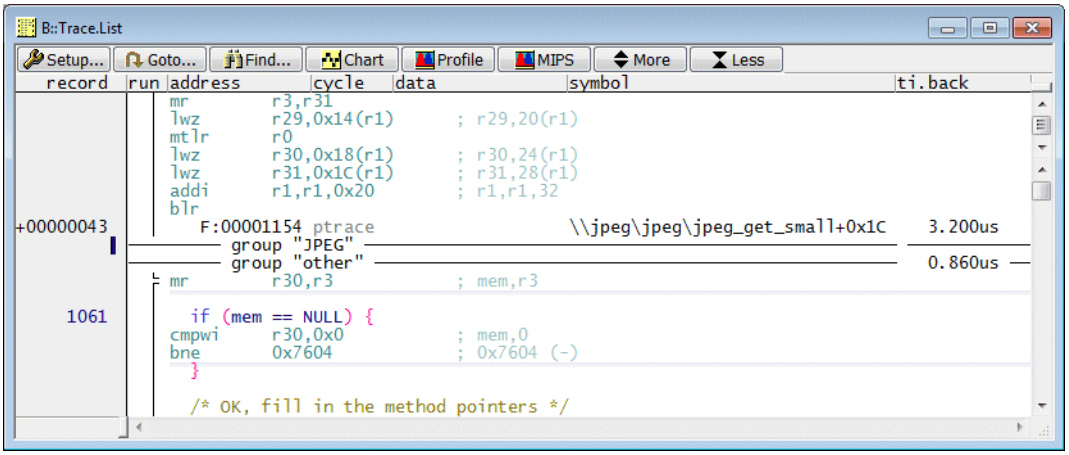




If a group is enabled, the following features are added by checking hide:



- The trace information recorded for the group members is hidden.



- The group represents its members in all trace analysis windows. No details about group members are displayed (same as merge).
- Step over group members during HLL single stepping.

Format:	**GROUP.COLOR** *<group_name> <color>*
<color>:	**NONE** **BLACK** **MAROON** **GREEN** **OLIVE** **NAVY** **PURPLE** **TEAL** **SILVER** **GREY** **RED** **LIME** **YELLOW** **BLUE** **FUCHSIA** **AQUA** **WHITE**

Defines the color that is used to mark the group members.

The following color convention are used:

RED	To mark the OS kernel.
YELLOW	To mark kernel drivers and libraries.
BLUE	To mark virtual machine byte code e.g. Android/Dalvik.
GREEN	To mark the application/application processes.

Example:

```
GROUP.COLOR "Layer 1" FUCHSIA    ; Specify color
```

See also

■ GROUP.Create ■ GROUP

▲ 'PowerView - Screen Display' in 'PowerView User's Guide'

Format:	**GROUP.Create** [*<group_name>* {*<group_member>*}] [**/***<option>*]
<group_member>:	*<address_range>* | *<function>* | *<module>* | *<program>* | *<source>*
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** *<color>*

The command **GROUP.Create** allows to create a new group. Group members can be defined by module name, function name, etc. Without options, the **GROUP.Create** dialog window is opened.

ENable (default)	Enable the GROUP features.
DISable	Disable the GROUP features.
SHOW (default)	Display the instructions of the GROUP members together with the GROUP indicator (COLOR).
HIDE	Suppress the display of the instructions of the GROUP members in the trace listing and step over the instructions of the GROUP members during HLL single stepping. The group represents its members in all trace analysis windows.
SEParate (default)	Display the measurement results separately for each group member if a trace analysis command is used.
Merge	The group represents its members in all trace analysis windows. No details about group members are displayed.
DIALOG	Deprecated.
<color>	Define the color for the GROUP indicator.

Examples:

```
GROUP.Create                          ; open GROUP.Create dialog window
```

```
GROUP.Create "kernel" \os_module1 \os_module2 \os_scheduler

GROUP.Create "Layer 1"   0x3F0000--0x3FA533 /LIME

GROUP.Create "INT" sYmbol.SECPRANGE(\.interrupt) /MAROON /HIDE
```

See also

- GROUP.COLOR
- GROUP.CreateFunctions
- GROUP.CreateLabels
- GROUP.CreateModules
- GROUP.CreatePrograms
- GROUP.CreateSources
- GROUP.CreateTASK
- GROUP
- GROUP.Delete
- GROUP.DeleteTASK
- GROUP.DISable
- GROUP.ENable
- GROUP.HIDE
- GROUP.List
- GROUP.Merge
- GROUP.RESet
- GROUP.SEParate
- GROUP.SHOW
- <trace>.Chart.GROUP
- <trace>.STATistic.GROUP

▲ 'Release Information' in 'Legacy Release History'

GROUP.CreateFunctions — Pool functions to group

Format:	**GROUP.CreateFunctions** *<group_name>* *<pattern>*|*<function>* [{**/***<option>*}]
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** **DIALOG** *<color>*

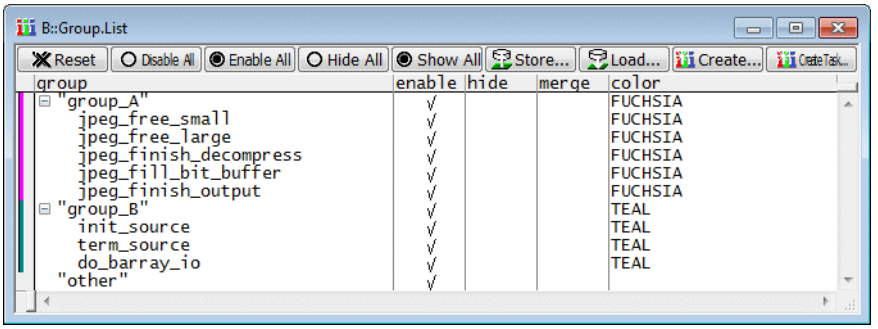
Pools the functions to groups.

<option> For a description of the options, refer to the **GROUP.Create** command.

Example:

```
; display symbol listing for all functions
sYmbol.List.Function

; pool all functions that match the specified name pattern to the
; group "group_A"
; assign color FUCHSIA to "group_A"
GROUP.CreateFunctions "group_A" jpeg_f* /FUCHSIA

; create group "group_B" that contains the function init_source
GROUP.CreateFunctions "group_B" init_source
```

```
; add function term_source to the group "group_B"
GROUP.CreateFunctions "group_B" term_source

; add function do_barray_io to the group "group_B"
; assign color TEAL to "group_B"
GROUP.CreateFunctions "group_B" do_barray_io /TEAL

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP

GROUP.CreateLabels — Use labels to pool address ranges to group

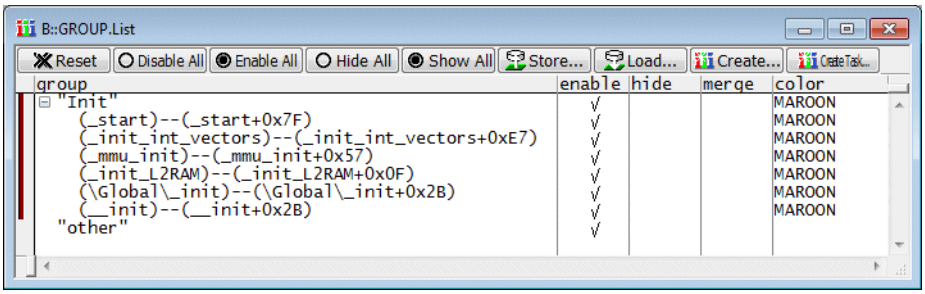
Format:	**GROUP.CreateLabels** *<group_name>* *<pattern>* | *<label>* [{/*<option>*}]
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** **DIALOG** *<color>*

Pools address ranges to groups. Each address range starts at a label and ends at the next label.

<option> For a description of the options, refer to the **GROUP.Create** command.

Example:

```
; pool all address ranges that start with a label of the specified name
; pattern to the group "Init"
GROUP.CreateLabels "Init" _*init*

; add address range that starts with label _start to the group "Init"
; assign color MAROON to the group "Init"
GROUP.CreateLabels "Init" _start /MAROON

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP

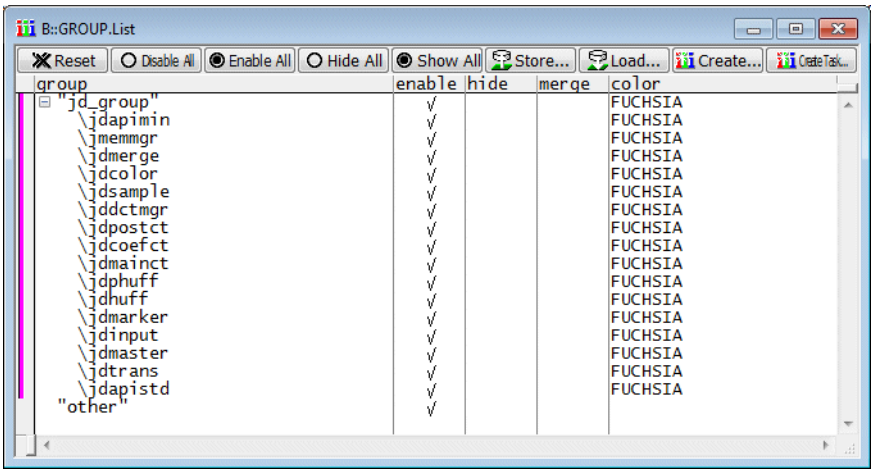
Format:	**GROUP.CreateModules** *<group_name>* *<pattern* | *module>* [{*/<option>*}]
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** *<color>*

Pools modules to group.

<option> For a description of the options, refer to the **GROUP.Create** command.

Example:

```
; display sYmbol listing for all functions
sYmbol.List.Module

; pool all modules that match the specified name pattern to the
; group "jd_group"
GROUP.CreateModules "jd_group" jd*

; add modules jmemmgr to group "jd_group"
; assign color FUCHSIA to group "jd_group"
GROUP.CreateModules "jd_group" jmemmgr /FUCHSIA

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP

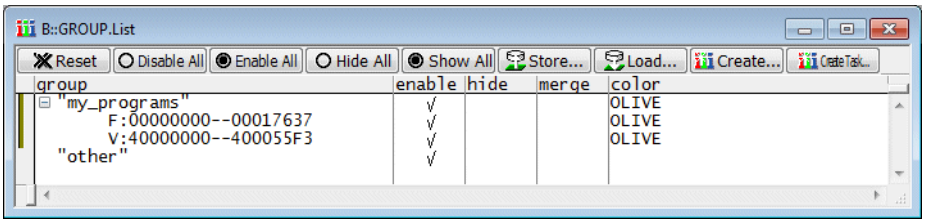
Format:	**GROUP.CreatePrograms** *<group_name>* *<pattern>*|*<program>* [{/*<option>*}]
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** *<color>*

Pools the programs that correspond to the specified name pattern to a new group.

<option> For a description of the options, refer to the **GROUP.Create** command.

Example:

```
; display symbol listing for all programs
sYmbol.List.Program

; pool all programs that match the specified name pattern to the
; group "my_programs"
GROUP.CreatePrograms "my_programs" j*

; add program im02_bf1x to group "my_programs"
; assign color OLIVE to group "my_programs"
GROUP.CreatePrograms "my_programs" im02_bf1x /OLIVE

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP

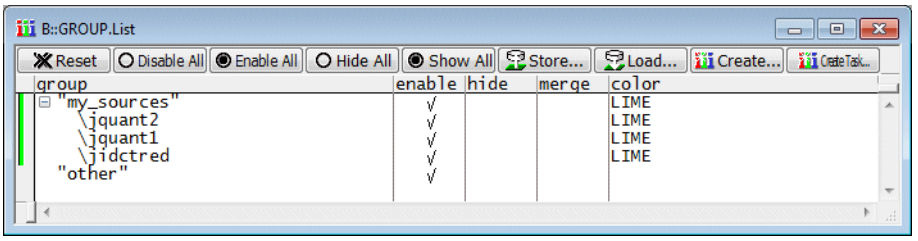
Format:	**GROUP.CreateSources** *<group_name>* *<pattern>*|*<source>* [{/*<option>*}]
<option>:	**ENable** | **DISable** **SHOW** | **HIDE** **SEParate** | **Merge** *<color>*

Pools the source files that correspond to the specified name pattern to a new group.

<option> For a description of the options, refer to the **GROUP.Create** command.

Example:

```
; display symbol listing for all sources
sYmbol.List.SOURCE

; pool all sources that match the specified name pattern to the
; group "my_sources"
GROUP.CreateSources "my_sources" *\mpc5xxx\mpc5646c_jpeg\jq*.c

; add all sources that match the specified name pattern to the group
; "my_sources"
; assign color LIME to group "my_sources"
GROUP.CreateSources "my_sources" *\mpc5xxx\mpc5646c_jpeg\ji*.c /LIME

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP

Format:	**GROUP.CreateTASK** *<group_name>* {*<task>*} [{/*<option>*}]
<task>:	*<task_magic>* | *<task_id>* | "*<task_name>*"
<option>:	**ENable** | **DISable** **SEParate** | **Merge** *<color>*

Pools tasks to a group. The grouping of tasks affects only the following commands:

Trace.STATistic.TASK	Display task activity statistic.
Trace.Chart.TASK	Display a task activity chart.
Trace.STATistic.TASKState	Display task state statistic.
Trace.Chart.TASKState	Display task state time chart.
Trace.PROfileChart.TASK	Display a task activity graph.
MIPS.STATistic.TASK	Display the MIPS per task numerically.
MIPS.PROfileChart.TASK	Display the MIPS per task graphically.

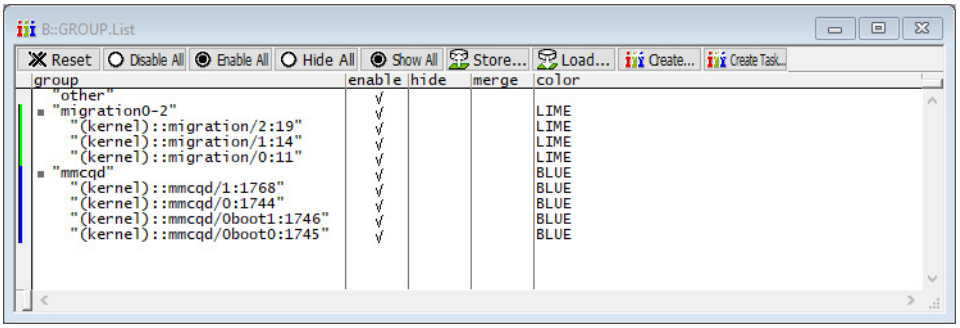
<option>	For a description of the options, refer to the **GROUP.Create** command.
<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).
<task_name>	This command supports task name with wildcard. If using wildcard in *task_name*, it will search the corresponding tasks and list them to the group.

Example for Linux:

B::TASK.List

magic	name	id	space	traceid	core	sel	stop
FFFF800012B05E80	kworker/7:0:46	46.	(kernel)	000000000000002E			◆
FFFF800012B06C00	kworker/7:0H:47	47.	(kernel)	000000000000002F			◆
FFFF80001218DE80	kworker/7:1:78	78.	(kernel)	000000000000004E			◆
FFFF800011975100	kworker/7:1H:1774	1774.	(kernel)	00000000000006EE			◆
FFFF8000129CB600	kworker/u16:0:5	5.	(kernel)	0000000000000005			◆
FFFF8000120E8D80	kworker/u16:1:50	50.	(kernel)	0000000000000032			◆
FFFF80001218C380	kworker/u16:2:59	59.	(kernel)	000000000000003B			◆
FFFF800012208000	kworker/u16:3:63	63.	(kernel)	000000000000003F			◆
FFFF80001220 8D80	kworker/u16:4:70	70.	(kernel)	0000000000000046			◆
FFFF800011CEC380	kworker/u16:5:435	435.	(kernel)	00000000000001B3			◆
FFFF8000110A3600	login:1806	1806.	login	000000000000070E			◆
FFFF8000110C9B00	machine_state_m:1845	1845.	machine_state_m	0000000000000735			◆
FFFF800011E20000	machine_state_m:2026	2026.	machine_state_m	00000000000007EA			◆
FFFF800011E20D80	machine_state_m:2027	2027.	machine_state_m	00000000000007EB			◆
FFFF800012A10D80	migration/0:11	11.	(kernel)	000000000000000B			◆
FFFF800012A41B00	migration/1:14	14.	(kernel)	000000000000000E			◆
FFFF800012A45E80	migration/2:19	19.	(kernel)	0000000000000013			◆
FFFF800012A7A880	migration/3:24	24.	(kernel)	0000000000000018			◆
FFFF800012A7EC00	migration/4:29	29.	(kernel)	000000000000001D			◆
FFFF800012ACB600	migration/5:34	34.	(kernel)	0000000000000022			◆
FFFF800012B00000	migration/6:39	39.	(kernel)	0000000000000027			◆
FFFF800012B04380	migration/7:44	44.	(kernel)	000000000000002C			◆
FFFF8000129CC380	mm_percpu_wq:6	6.	(kernel)	0000000000000006			◆
FFFF800011D92880	mmcqd/0:1744	1744.	(kernel)	00000000000006D0			◆
FFFF800011D95E80	mmcqd/0boot0:1745	1745.	(kernel)	00000000000006D1			◆
FFFF8000110A0000	mmcqd/0boot1:1746	1746.	(kernel)	00000000000006D2			◆
FFFF80001183B600	mmcqd/1:1768	1768.	(kernel)	00000000000006E8			◆
0000000000000838	modprobe:2104	2104.	modprobe				◆
FFFF8000120E8000	netns:49	49.	(kernel)	0000000000000031			◆
FFFF800011935100	nfsiod:911	911.	(kernel)	000000000000038F			◆

```
; display task list
TASK.List

; pool specified tasks to group "migration0-2"
; use <task_name> to specify tasks
; assign color LIME to group "migration0-2"
GROUP.CreateTASK "migration0-2" "migration/0:11" "migration/1:14" \
                                                  "migration/2:19" /LIME

; pool specified tasks to group "mmcgd"
; use <task_name> to specify tasks
; assign color BLUE to group "mmcgd"
GROUP.CreateTASK "mmcgd" "mmcgd*" /BLUE

; pool specified tasks to group "migration0-2"
; use <task_magic> to specify tasks
; assign color LIME to group "migration0-2"
GROUP.CreateTASK "migration0-2" 0xFFFF800012A10D80 0xFFFF800012A41B00 \
                                                    0xFFFF800012A45E80 /LIME

; pool specified tasks to group "migration0-2"
; use <task_id> to specify tasks
; assign color LIME to group "migration0-2"
GROUP.CreateTASK "migration0-2" 11. 14. 19. /LIME

; list group definition
GROUP.List
```



See also

- GROUP.Create
- GROUP
- GROUP.DeleteTASK

GROUP.Delete — Delete the specified group

Format:	**GROUP.Delete** [*<group_name>* | *<range>* | *<address>*]

Deletes the specified GROUP. If no group is specified, then all GROUPs are deleted.

Example:

```
GROUP.Delete "kernel"                  ; delete the "kernel" group

GROUP.Delete 0x3F0000--0x3FA533        ; delete group in the address range
```

See also

- GROUP
- GROUP.Create

GROUP.DeleteTASK — Delete specified task from group

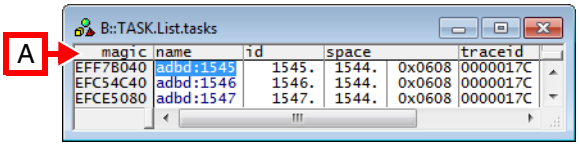
Format:	**GROUP.DeleteTASK** [*<task_magic>* | *<task_id>* | "*<task_name>*"]

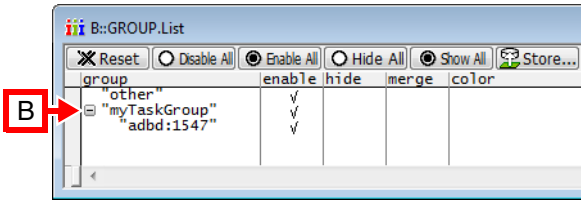
Deletes the specified task from a group of tasks based on the task's magic number, ID, or name. If no group is specified, then all GROUPs are deleted.

<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

Example:

```
TASK.List.tasks      ;list all task names including their magic numbers
                     ;and IDs

GROUP.List           ;display an overview of all groups

;create a task group named 'myTaskGroup' and add three tasks to it
GROUP.CreateTASK   "myTaskGroup" "adbd:1545" "adbd:1546" "adbd:1547"

;for demo purposes, let's delete two tasks based on magic number and ID
GROUP.DeleteTASK   0xEFF7B040      ;magic number of task
GROUP.DeleteTASK   1546.           ;ID of task
```





A The magic numbers, names and IDs of the tasks are displayed in the **TASK.List.tasks** window.

B Result: Two of the three tasks have been deleted from the group named ‘myTaskGroup’.

See also

- GROUP
- GROUP.Create
- GROUP.CreateTASK

GROUP.DISable — Disable a group

Format:	**GROUP.DISable** [*<group_name>* | *<range>* | *<address>*]

Disables a group.

```
GROUP.DISable "kernel"

GROUP.DISable 0x3F0000--0x3FA533
```

See also

- GROUP
- GROUP.Create

GROUP.ENable — Enable a group

Format:	**GROUP.ENable** [*<group_name>* | *<range>* | *<address>*]

Enables a group. For details, refer to **Features**.

Examples:

```
GROUP.ENable "kernel"

GROUP.ENable 0x3F0000--0x3FA533
```

See also

- GROUP
- GROUP.Create

GROUP.HIDE — Hide group from debugging

Format:	**GROUP.HIDE** [*<group_name>* | *<range>* | *<address>*]

Hides a group. For details, refer to **Features.**

Example:

```
GROUP.HIDE "kernel"

Trace.List

GROUP.SHOW "kernel"
```

See also

- GROUP
- GROUP.Create

▲ 'Release Information' in 'Legacy Release History'

GROUP.List — List all specified groups

Format:	**GROUP.List**

Displays all group definitions.

See also

- GROUP
- GROUP.Create

GROUP.Merge — Merge group members in statistic

Format:	**GROUP.Merge** *<name>*

Merges group members in all trace analysis windows. For details, refer to **Features**.

Example:

```
GROUP.Merge "layer 1"

Trace.STATistic.Func

GROUP.SEParate "layer 1"
```

See also

- GROUP
- GROUP.Create

GROUP.RESet — Clear all group specifications

Format:	**GROUP.RESet**

Resets all group settings to default.

Example:

```
GROUP.RESet
```

See also

- GROUP
- GROUP.Create

GROUP.SEParate — Separate group members in statistic

Format:	**GROUP.SEParate** *<name>*

Displays details about group members in all trace analysis windows (default). For details, refer to **Features.**

Example:

```
GROUP.SEParate "layer 1"

Trace.STATistic.Func

GROUP.Merge "layer 1"
```

See also

- GROUP
- GROUP.Create

Format:	**GROUP.SHOW** [*<group_name>* | *<range>* | *<address>*]

Shows a group. For details, refer to **Features.**

Example:

```
GROUP.SHOW "kernel"

Trace.List

GROUP.HIDE "kernel"
```

See also

- GROUP
- GROUP.Create