LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual VxWorks

Release 02.2024

MANUAL

OS Awareness Manual VxWorks

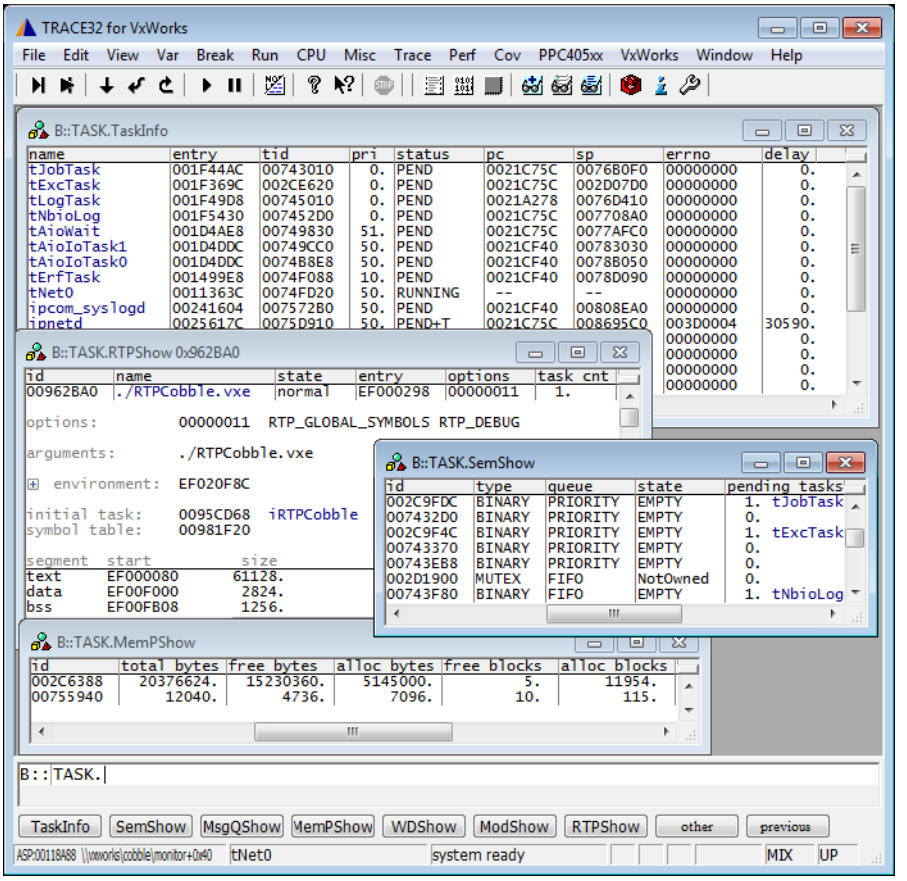
OS Awareness Manual VxWorks

Version 04-Mar-2024

History

04-Feb-21 Removing legacy command TASK.TASKState.

Overview



The OS Awareness for VxWorks contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Training Basic Debugging"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

Supported Versions

Currently VxWorks is supported for the following versions:

- VxWorks 5.x, 6.x and 7 on several architectures.
- VxWorks 653 2.x on several architectures

Please note that VxWorks 653 3.x is based on Wind River Hypervisor, and supported by the Awareness for Wind River Hypervisor. Please use the awareness and scripts from the directory "~~/demo/*<arch>*/kernel/vxworks653".

Configuration

The **TASK.CONFIG** command loads an extension definition file called “vxworks.t32” (directory “~~/demo/*<arch>*/kernel/vxworks”). It contains all necessary extensions.

TASK.CONFIG ~~/demo/*<arch>*/kernel/vxworks/vxworks.t32

This command:

- Configures the OS Awareness for VxWorks,
- Loads an additional VxWorks menu (see “**VxWorks Specific Menu**”),
- Sets the default stack pattern to 0xEE (see “**Task Stack Coverage**”),
- Configures the symbol autoloader (see “**Symbol Autoloader**”).

The OS Awareness tries to locate all needed VxWorks internals automatically. For this purpose all symbol information of the kernel application must be loaded and accessible at any time the OS Awareness is used. See also **Hooks & Internals**.

It is recommended to load the awareness first, after the kernel is initialized, to avoid faulty memory accesses by the OS Awareness. A good place would be at or after usrRoot().

If the application enables the MMU and/or uses RTPs, configure the **MMU Support**, after the kernel initialized the MMU (i.e. after usrMmuInit(), e.g. at usrAppInit()).

If you want to display the OS objects “On The Fly” while the target is running, you need to have access to memory while the target is running. In case of ICD, you have to enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent).

The extension definition file will refer to further files in the extension directory. Be sure that the installation in this directory is complete.

If, for any reason, you have to specify VxWorks internal addresses manually, contact LAUTERBACH for assistance.

See also the example script “~~/demo/*<arch>*/kernel/vxworks/vxworks.cmm”

Quick Configuration Guide

To get a quick access to the features of the OS Awareness for VxWorks with your application, follow this roadmap:

1. Start the TRACE32 Debugger.
2. Load your application as normal.
3. Wait until the kernel is initialized.
4. Load the VxWorks extension file:

```
TASK.CONFIG ~~/demo/<arch>/kernel/vxworks/vworks.t32
```

5. Start your application.
6. If the application enables the MMU and/or uses RTPs, configure the **MMU Support**, after the kernel initialized the MMU (i.e. after usrMmuInit(), e.g. at usrAppInit()).

Now you can access the VxWorks extensions through the menu.

See also the example script "~~/demo/*<arch>*/kernel/vxworks/vxworks.cmm"

Hooks & Internals in VxWorks

No hooks are used in VxWorks.

For retrieving the kernel data and structures, the OS Awareness uses the global kernel symbols and structure definitions. Ensure that access to those structures is possible every time when features of the OS Awareness are used.

Be sure that your application is compiled and linked with debugging symbols switched on.

VxWorks 653 v2.x:

In some BSPs, the clock interrupt handler `sysClkInt()` checks for missed timer ticks and raises an error if some ticks were lost (e.g. i8253Timer.c, ppcDecTimer.c). This check could be triggered, if the target is halted with the debugger. To skip this check after a break, set the variable "ignoreLostTicks" each time before the execution is resumed. You can automate this with these **Data.PROLOG** settings:

```
Data.PROLOG.SEQuence SET ignoreLostTicks 1
Data.PROLOG.ON
```

In BSPs for PowerPC e500 cores, `sysToMonitor()` (sysLib.c) sets the MSR register to zero. This disables JTAG debugging, too. If you want to debug beyond this point, please change or patch this to keep the MSR:DE bit set:
`vxMsrSet(0x200);`

Features

The OS Awareness for VxWorks supports the following features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. The following information can be displayed:

TASK.TaskInfo	Tasks
TASK.WDShow	Watchdogs
TASK.MsgQShow	Message queues
TASK.SemShow	Semaphores
TASK.MemPShow	Memory partitions
TASK.ModShow	Modules
TASK.RTPShow	RTPs
TASK.PDShow	Protection domains
TASK.LKUP	Target symbol database

For a description of the commands, refer to chapter “**VxWorks Commands**”.

If your hardware allows memory access while the target is running, these resources can be displayed “On The Fly”, i.e. while the application is running, without any intrusion to the application.

Without this capability, the information will only be displayed if the target application is stopped.

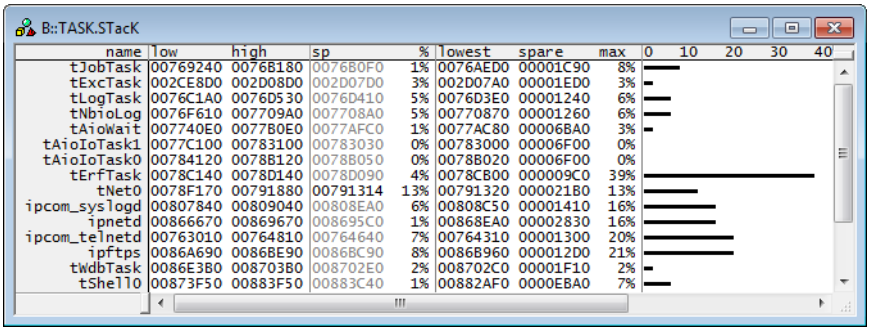
Task Stack Coverage

For stack usage coverage of tasks, you can use the **TASK.STacK** command. Without any parameter, this command will open a window displaying with all active tasks. If you specify only a task magic number as parameter, the stack area of this task will be automatically calculated.

To use the calculation of the maximum stack usage, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is zero).

To add/remove one task to/from the task stack coverage, you can either call the **TASK.STacK.ADD** or **TASK.STacK.ReMove** commands with the task magic number as the parameter, or omit the parameter and select the task from the **TASK.STacK.*** window.

It is recommended to display only the tasks you are interested in because the evaluation of the used stack space is very time consuming and slows down the debugger display.



B::TASK.STacK

name	low	high	sp	%	lowest	spare	max
tJobTask	00769240	0076B180	0076B0F0	1%	0076AED0	00001C90	8%
tExcTask	002CE8D0	002D08D0	002D07D0	3%	002D07A0	00001ED0	3%
tLogTask	0076C1A0	0076D530	0076D410	5%	0076D3E0	00001240	6%
tNbioLog	0076F610	007709A0	007708A0	5%	00770870	00001260	6%
tAioWait	007740E0	0077B0E0	0077AFC0	1%	0077AC80	00006BA0	3%
tAioIoTask1	0077C100	00783100	00783030	0%	00783000	00006F00	0%
tAioIoTask0	00784120	0078B120	0078B050	0%	0078B020	00006F00	0%
tErfTask	0078C140	0078D140	0078D090	4%	0078CB00	000009C0	39%
tNet0	0078F170	00791880	00791314	13%	00791320	000021B0	13%
ipcom_syslogd	00807840	00809040	00808EA0	6%	00808C50	00001410	16%
ipnetd	00866670	00869670	008695C0	1%	00868EA0	00002830	16%
ipcom_telnetd	00763010	00764810	00764640	7%	00764310	00001300	20%
ipftps	0086A690	0086BE90	0086BC90	8%	0086B960	000012D0	21%
tWdbTask	0086E3B0	008703B0	008702E0	2%	008702C0	00001F10	2%
tShell0	00873F50	00883F50	00883C40	1%	00882AF0	0000EBA0	7%

VxWorks typically initializes task stacks with the value 0xEE. When configuring the OS Awareness for VxWorks, this pattern is set for stack detection. If VxWorks uses a different pattern, inform the debugger about this with the command **TASK.STacK.PATtern**.

Task-Related Breakpoints

Any breakpoint set in the debugger can be restricted to fire only if a specific task hits that breakpoint. This is especially useful when debugging code which is shared between several tasks. To set a task-related breakpoint, use the command:

Break.Set *<address>*|*<range>* [**/***<option>*] **/TASK** *<task>*	Set task-related breakpoint.

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- For a general description of the **Break.Set** command, please see its documentation.

By default, the task-related breakpoint will be implemented by a conditional breakpoint inside the debugger. This means that the target will *always* halt at that breakpoint, but the debugger immediately resumes execution if the current running task is not equal to the specified task.

NOTE:	Task-related breakpoints impact the real-time behavior of the application.

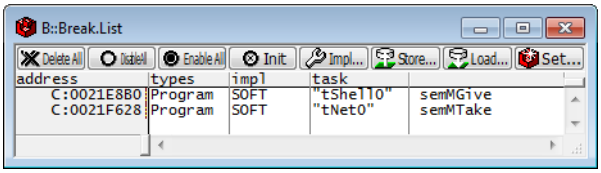
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering the tasks.

For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

Break.CONFIG.UseContextID ON	Enables the comparison to the whole Context ID register.
Break.CONFIG.MatchASID ON	Enables the comparison to the ASID part only.
TASK.List.tasks	If **TASK.List.tasks** provides a trace ID (**traceid** column), the debugger will use this ID for comparison. Without the trace ID, it uses the magic number (**magic** column) for comparison.

When single stepping, the debugger halts at the next instruction, regardless of which task hits this breakpoint. When debugging shared code, stepping over an OS function may cause a task switch and coming back to the same place - but with a different task. If you want to restrict debugging to the current task, you can set up the debugger with **SETUP.StepWithinTask ON** to use task-related breakpoints for single stepping. In this case, single stepping will always stay within the current task. Other tasks using the same code will not be halted on these breakpoints.

If you want to halt program execution as soon as a specific task is scheduled to run by the OS, you can use the **Break.SetTask** command.



Task Context Display

You can switch the whole viewing context to a task that is currently not being executed. This means that all register and stack-related information displayed, e.g. in **Register**, **Data.List**, **Frame** etc. windows, will refer to this task. Be aware that this is only for displaying information. When you continue debugging the application (**Step** or **Go**), the debugger will switch back to the current context.

To display a specific task context, use the command:

Frame.TASK [*<task>*]	Display task context.

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

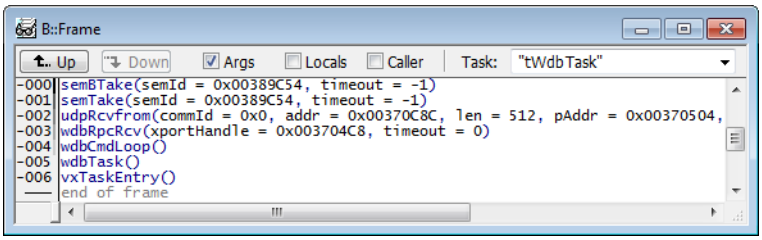
To display the call stack of a specific task, use the following command:

Frame /Task *<task>*	Display call stack of a task.

If you'd like to see the application code where the task was preempted, then take these steps:

1. Open the **Frame /Caller /Task** *<task>* window.

2. Double-click the line showing the OS service call.



MMU Support

To provide full debugging possibilities on MMU enabled systems, the debugger has to know, how virtual addresses are translated to physical addresses and vice versa. All MMU and TRANSlation commands refer to this necessity. If the VxWorks configuration variable "`_WRS_CONFIG_MMULESS_KERNEL`" is *not* set, the MMU will be activated by the kernel.

Space IDs

RTPs of VxWorks may reside virtually on the same address. To distinguish those addresses, the debugger uses an additional space ID that specifies to which virtual memory space the address refers. The command **SYStem.Option.MMUSPACES ON** enables the additional space ID. For all tasks using the kernel address space, the space ID is zero. For tasks within an RTP, the debugger assigns a space ID unique to the RTP.

See also chapter "**Debugging Real Time Processes**".

MMU Declaration

To access the virtual and physical addresses correctly, the debugger needs to know the format of the MMU tables in the target.

The following command is used to declare the basic format of MMU tables:

MMU.FORMAT *<format>* [*<base_address>* [*<logical_kernel_address_range>* *<physical_kernel_address>*]]	Define MMU table structure

The MMU will be initialized first in usrMmuInit(). In order to read out the correct information, the MMU declaration must happen *after* this function ran, e.g. when reached usrAppInit(). To get the kernel TTB address, the debugger needs to read the target memory. If this is not possible while the target runs, halt the target before declaring the MMU.

<format> Options for ARM:

<format>	Description
STD	Standard format defined by the CPU
TINY	MMU format using a tiny page size of only 1024 bytes

<format> Options for PowerPC:

<format>	Description
STD	Standard format defined by the CPU
VX653	MMU format for VXWORKS 653
VXWORKS.E500	VxWorks specific format for PowerPC e500 core with 128-bit PTEs
VXWORKS.E500MC	VxWorks specific format for PowerPC e500mc core with 36 bit physical addresses (PPC64 only)with 128-bit PTEs
VXWORKS.E500_64	VxWorks specific format for PowerPC e500 core (PPC64 only)with 128-bit PTEs
VXWORKS.E6500	VxWorks specific format for PowerPC e6500 core with 64-bit PTEs

<format> Options for RISC-V:

<format>	Description
STD	Automatic detection of the page table format from the SATP register.
SV32	32-bit page table format (for SV32 targets only)
SV39	39-bit page table format (for SV64 targets only)
SV48	48-bit page table format (for SV64 targets only)

<format> Options for x86:

<format>	Description
EPT	Extended page table format (type autodetected)
EPT4L	Extended page table format (4-level page table)
EPT5L	Extended page table format (5-level page table)
P32	32-bit format with 2 page table levels
PAE	Format with 3 page table levels
PAE64	64-bit format with 4 page table levels
PAE64L5	64-bit format with 5 page table levels
STD	Automatic detection of the page table format used by the CPU

<base_address>

<base_address> specifies the TTB of the kernel. Use **TASK.RTP.TTB(0)**. Take care that the kernel TTB can only be read, if the target is halted.

<logical_kernel_address_range>

<logical_kernel_address_range> specifies the virtual address range of the kernel.
For ARM, PowerPC and x64 architecture, this is typically specified in adrSpaceArchLibP.h, as `KERNEL_SYS_MEM_RGN_BASE` and `KERNEL_SYS_MEM_RGN_SIZE`. For x86, the start address is zero, and the size is typically the size of RAM. The default values are:

ARM	`0x0--0x1fffffff`
PowerPC	`0x0--0x1fffffff`
x64	`0xffffffff80000000--0xffffffffffffffff`
x86	`0x0--<LOCAL_MEM_SIZE+LOCAL_MEM_LOCAL_ADRS>`

<physical_kernel_address>

<physical_kernel_address> specifies the physical start address of the kernel.

The kernel code, which resides in the kernel space, can be accessed by any RTP, regardless of the current space ID. Use the command **TRANSlation.COMMON** to define the complete address range that is addressed by the kernel as commonly used area. In ARM PowerPC and x86 architectures, this is the same range as the kernel range specified with **TRANSlation.COMMON**. In x64 architectures, specify a bigger range as mentioned in the example below.

Enable the debugger's table walk with **TRANSlation.TableWalk ON**, and switch on the debugger's MMU translation with **TRANSlation.ON**.

Example for ARM with RAM at physical address 0x10000000;

```
IF RUN()
  BREAK
MMU.FORMAT STD task.rtp.ttb(0) 0x0--0x1fffffff 0x10000000
TRANSlation.COMMON 0x0--0x1fffffff
TRANSlation.TableWalk ON
TRANSlation.ON
```

Example for PowerPC with e500 core:

```
IF RUN()
  BREAK
MMU.FORMAT VXWORKS.E500 task.rtp.ttb(0) 0x0--0x1fffffff 0x0
TRANSlation.COMMON 0x0--0x1fffffff
TRANSlation.TableWalk ON
TRANSlation.ON
```

Example for x86 with 1GB RAM:

```
IF RUN()
  BREAK
MMU.FORMAT STD A:task.rtp.ttb(0) 0x0--0x3fffffff 0x0
TRANSlation.COMMON 0x0--0x3fffffff
TRANSlation.TableWalk ON
TRANSlation.ON
```

Example for x64 with 1GB RAM. Please see also the sample scripts in the ~~/demo directory.

```
IF RUN()
  BREAK
MMU.FORMAT STD A:task.rtp.ttb(0) 0xffffffff80000000--0xffffffffffffffff 0x0
TRANSlation.COMMON 0xffff800000000000--0xffffffffffffffff
TRANSlation.TableWalk ON
TRANSlation.ON
```

Symbol Autoloader

The OS Awareness for VxWorks contains a symbol autoloader which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding Vxworks kernel modules, RTPs and libraries, and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader, the debugger invokes the appropriate command. The command is usually a call to a PRACTICE script that loads the symbol file to the appropriate addresses.

The command **sYmbol.AutoLOAD.List** shows a list of all known address ranges/components and their symbol load commands.

The autoloader reads the target's tables for modules RTPs and libraries and fills the autoloader list with the modules found on the target. All necessary information, such as load addresses, are retrieved from kernel-internal information.

The symbol autoloader is set automatically when configuring the OS Awareness for VxWorks. If, for any reason, you need to change the behavior of the symbol autoloader, use:

sYmbol.AutoLOAD.CHECKCoMmanD "*<action>*"

<action>	Action to take for symbol load, usually: "`DO ~~/demo/arm/kernel/vxworks/autoload.cmm` "

If an address is accessed that is covered by the autoloader list, the autoloader calls *<action>* and appends the load addresses of the component to the action. Usually, *<action>* is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example script "autoload.cmm" in the ~~/demo directory.

The point in time when the component information is retrieved from the target can be set:

sYmbol.AutoLOAD.CHECK [**ON** | **OFF** | **ONGO**]

(no argument)	A single **sYmbol.AutoLOAD.CHECK** command refreshes the information about the target.
ON	The debugger automatically reads the information on every go/halt or step cycle. This significantly slows down the debugger's speed when single stepping.
ONGO	The debugger automatically reads the information on every go/halt cycle, but not when single stepping.
OFF	no automatic update of the autoloader table will be done, you have to manually trigger the information read when necessary. To accomplish that, execute the **sYmbol.AutoLOAD.CHECK** command without arguments.

NOTE:	The autoloader covers only components that are already started. Components that are not in the current module table are not covered.

SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application runs may change over time.

To support such SMP systems, the debugger allows a "system view", where one TRACE32 PowerView GUI is used for the whole system, i.e. for all cores that are used by the SMP OS. For information about how to set up the debugger with SMP support, please refer to the **Processor Architecture Manuals**.

All core relevant windows (e.g. **Register.view**) show the information of the current core. The state line of the debugger indicates the current core. You can switch the core view with the **CORE.select** command.

Target breaks, be they manual breaks or halting at a breakpoint, halt all cores synchronously. Similarly, a **Go** command starts all cores synchronously. When halting at a breakpoint, the debugger automatically switches the view to the core that hit the breakpoint.

Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

Debugging Modules

If you want to debug kernel modules that are dynamically loaded within VxWorks, you have to load the symbols into the debugger. The symbols need to be relocated to the actual addresses, where VxWorks loaded the module.

Check **TASK.ModShow**, if the module is shown in the module list.

NOTE:	Loading the symbols of a module *only* works, if the debugger has access to memory to read out the relocation addresses.

Use the **Symbol Autoloader** to load the symbols of a module:

```
; specify the name of the module
sYmbol.AutoLOAD.TOUCH "mymod.out"
```

If the symbol autoloader is configured, you can use the local menu in **TASK.ModShow** to load the symbols: Right-click on the module's ID or name, and then select **Load Module Symbols**.

Alternatively, you can load the symbols of a module manually.
Use the "`/RELOCTYPE 2`" load option to relocate the symbols to the appropriate addresses.
Use the "`/NoCODE`" option to load only the symbols and use the "`/NoClear`" option to keep the VxWorks kernel symbols.

The following example script loads the symbols of a module called "mymod.out":

```
; load the symbols of mymod.out
Data.LOAD.Elf mymod.out /NoCODE /NoClear /RELOCTYPE 2
```

Debugging Real Time Processes

If you want to debug real time processes (RTPs) that are dynamically loaded within VxWorks, you have to load the symbols into the debugger. The symbols need to be relocated to the actual addresses, where VxWorks loaded the RTP.

Check **TASK.RTPShow** , if the RTP is shown in the RTP list.

NOTE:	• Loading the symbols of an RTP *only* works, if the debugger has access to memory to read out the relocation addresses. • RTPs run in an MMU mapped virtual address range. The **MMU Support** with space IDs must be enabled to correctly support debugging RTPs.

Use the **symbol autoloader** to load the symbols of an RTP:

```
; specify the name of the RTP
sYmbol.AutoLOAD.TOUCH "myrtp.vxe"
```

I the symbol autoloader is configured, you can use the local menu in **TASK.RTPShow** to load the symbols: Right click on the RTP's ID or name and select "Load RTP Symbols".

Alternatively, you can load the symbols of an RTP manually.
Use the "`/LOCATEAT`" load option to relocate the symbols to the start address of the RTP.
Use the "`/NoCODE`" option to load only the symbols and use the "`/NoClear`" option to keep the VxWorks kernel symbols.

The following example script loads the symbols of an RTP called "myrtp.vxe":

```
; declare local variables
LOCAL &rtpid &spaceid &text

; get the space ID and load address
&rtpid=task.rtp.id("./myrtp.vxe")
&spaceid=task.rtp.spaceid(&rtpid)
&text=task.rtp.segaddr(".text",&rtpid)

; load the symbols of myrtp.vxe
Data.LOAD.Elf myrtp.vxe &spaceid:0 /NoCODE /NoClear /LOCATEAT &text
```

Debugging a Real Time Process from its entry point

If you want to debug your RTP from its entry point, you need to split the loading and starting of the RTP.

First, load the RTP (e.g. "myRtp.vxe") in the VxWorks command shell with the -s option, to keep the RTP in stopped state:

```
[vxWorks *]# rtp exec -s myRtp.vxe &
```

Then load the symbols of the RTP into the debugger, and set a breakpoint on its entry point. E.g.:

```
Break
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "myRtp.vxe"
Break.Set \\myRtp\main
Go
```

At last, start the RTP in the VxWorks command shell with its RTP ID, e.g. 0x12345678 (check with rtp list):

```
[vxWorks *]# rtp list
[vxWorks *]# rtp continue 0x12345678
```

Debugging Protection Domains

Protection domains (aka ARINC653 partitions) reside on a prelinked virtual address. All PDs use the same virtual address range (but of course different physical addresses). The MMU takes care to remap the virtual address range on a domain change.

The debugger needs to distinguish the different domain translations, to uniquely access a specific virtual address. For this, the debugger extends the virtual address by a “space ID”, which is a 16bit extension of the 32bit vitual address. Use the command **SYStem.Option.MMUSPACES ON** to switch on the address extension (space ID).

Load the symbols of your applications into the space ID of the PD. Use **TASK.PDShow** to see the space ID that belongs to your application. Use the “`/NoCODE`” option to load only the symbols and use the “`/NoClear`” option to keep the VxWorks kernel symbols. Note: do **not** load the symbols of the vxSysLib.sm; it may spoil the VxWorks Awareness.

Set up the debugger address translation to get access to each PD. Set **TRANSlation.COMMON** to the kernel area (everything below the partition virtual address). After VxWorks came up, scan the MMU tables o each partition with **TASK.MMU.SCANSPACE**. Switch on the debugger address translation with **TRANSlation.ON**.

Check **TASK.PDShow**, if the protection domain is shown in the PD list.

The following example script loads the symbols of two partitions”:

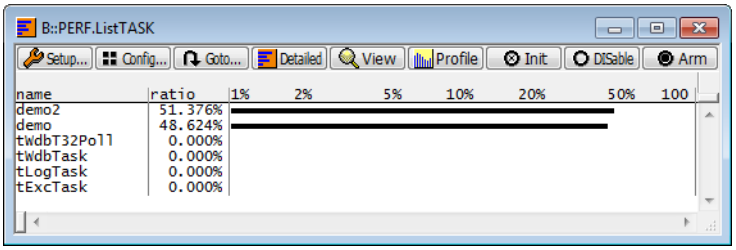
```
; declare local variables
local &virt
; reset symbols for MMUSPACES option
sYmbol.RESet
SYStem.Option.MMUSPACES ON

; load core OS symbols (needed by Awareness!)
Data.LOAD.Elf coreOS.sm /NoCODE

; load symbols of partitions
; lookup the space ID of each partition in TASK.PDShow
Data.LOAD.Elf myFirstPartition.sm  1:0 /NoCODE /NoClear
Data.LOAD.Elf mySecondPartition.sm 2:0 /NoCODE /NoClear

; Lookup "partitionVirtualAddress" in the "CoreOSDescription"
; of your XML file
&virt=0x40000000

; set up COMMON area everything below partition address
TRANSLATION.COMMON 0x0--(&virt-1)

; scan the MMU translation of each partition ID
TASK.MMU.SCANSPACE 1 &virt
TASK.MMU.SCANSPACE 2 &virt

; clean up translation table and switch on debugger translation
TRANSlation.CLEANUP
TRANSlation.ON
```

Dynamic Task Performance Measurement

The debugger can execute a dynamic performance measurement by evaluating the current running task in changing time intervals. Start the measurement with the commands **PERF.Mode TASK** and **PERF.Arm**, and view the contents with **PERF.ListTASK**. The evaluation is done by reading the ‘magic’ location (= current running task) in memory. This memory read may be non-intrusive or intrusive, depending on the **PERF.METHOD** used.

If **PERF** collects the PC for function profiling of processes in MMU-based operating systems (**SYStem.Option.MMUSPACES ON**), then you need to set **PERF.MMUSPACES**, too.

For a general description of the **PERF** command group, refer to **“General Commands Reference Guide P”** (general_ref_p.pdf).



Task Runtime Statistics

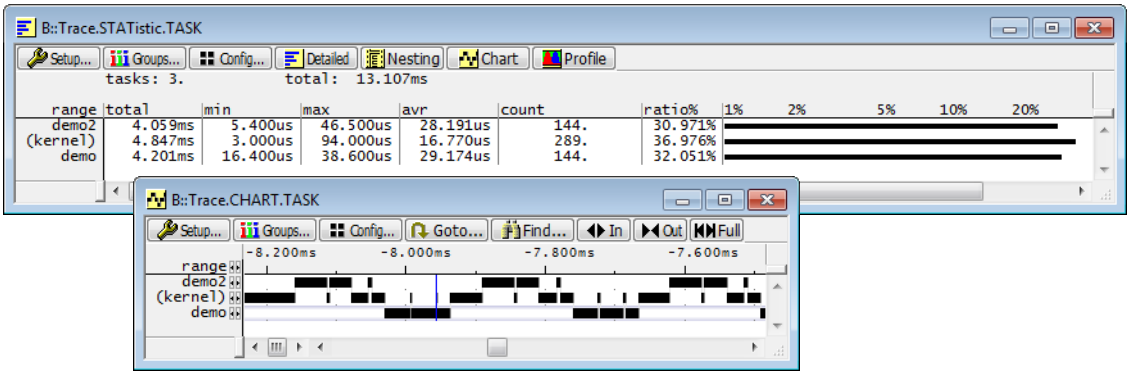
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **Logger**). For details, refer to **"OS-aware Tracing"** (glossary.pdf).

Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

To evaluate the contents of the trace buffer, use these commands:

Trace.List List.TASK DEFault	Display trace buffer and task switches
Trace.STATistic.TASK	Display task runtime statistic evaluation
Trace.Chart.TASK	Display task runtime timechart
Trace.PROfileSTATistic.TASK	Display task runtime within fixed time intervals statistically
Trace.PROfileChart.TASK	Display task runtime within fixed time intervals as colored graph
Trace.FindAll Address TASK.CONFIG(magic)	Display all data access records to the "magic" location
Trace.FindAll CYcle owner OR CYcle context	Display all context ID records

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".



Task Trace with ARM ContextID

On ARM architectures, VxWorks serves the ContextID register with the address space ID (ASID) of the RTP. This allows tracking the program flow of the kernel and RTPs and evaluation of the RTP switches. But it does not provide performance information of tasks.

To allow a detailed performance analysis on VxWorks tasks, the context ID must contain the task ID. Set the lower 8 bit of the context ID register with the RTPs ASID, and set the upper 24 bit with the ID of the task, i.e. "`(taskid << 8) | ASID`".

The VxWorks awareness needs to be informed about the changed format of the context ID:

TASK.Option THRCTX ON

To implement the above context ID setting, you need a VxWorks 7 version where the os_arch_arm component release number is at least 1.1.3.3. If you're using an older VxWorks version, contact Lauterbach for a patch.

In the VxWorks 7 Source Build Project, check for the option "PROCID_IN_CONTEXTIDR". If this option is available, set it to "yes".

In your application, implement a task switch hook to serve the PROCID field in the ContextID register. Implement a new assembly source file, e.g. ctxIdTrace.s.

Example for 32bit Arm systems:

```
#define _ASMLANGUAGE
#include <vxWorks.h>
#include <asm.h>
#include <prjParams.h>
FUNC_EXPORT(storeContextID)
FUNC_BEGIN(storeContextID)
    /* write new TCB pointer to Proc field in Context ID register */
    lsl   r1, r1, #0x8                /* pNewTcb <<= 8 */
    mrc   CP_MMU, 0, r2, c13, c0, 1   /* read Context ID register */
    and   r2, r2, #0xff               /* mask ASID */
    orr   r2, r2, r1                  /* add new proc ID */
    mcr   CP_MMU, 0, r2, c13, c0, 1   /* set new Context ID */
    bx    r14
FUNC_END(storeContextID)
```

Example for 64bit Arm systems:

```
#define _ASMLANGUAGE
#include <vxWorks.h>
#include <asm.h>
#include <prjParams.h>
FUNC_EXPORT(storeContextID)
FUNC_BEGIN(storeContextID)
    /* write new TCB pointer to Proc field in Context ID register */
    lsl   x1, x1, #0x8                /* pNewTcb <<= 8 */
    mrs   x2, CONTEXTIDR_EL1          /* read Context ID register */
    and   x2, x2, #0xff               /* mask ASID */
    orr   x2, x2, x1                  /* add new proc ID */
    msr   CONTEXTIDR_EL1, x2          /* set new Context ID */
    ret
FUNC_END(storeContextID)
```

Insert a task switch hook within your application

```
void storeContextID (WIND_TCB *pOldTcb, WIND_TCB *pNewTcb);
taskSwitchHookAdd((FUNCPTR)storeContextID);
```

Task Trace with PowerPC NPIDR

If the used PowerPC architecture supports the NPIDR register, you may use this register to trace task switches. Implement a task switch hook to serve the NPIDR register:

Implement a new assembly source file, e.g. pidTrace.s:

```
#define _ASMLANGUAGE
#include <vxWorks.h>
#include <asm.h>
#include <prjParams.h>
FUNC_EXPORT(storeNPIDR)
FUNC_BEGIN(storeNPIDR)
    /* write new TCB pointer to NPIDR register */
    mtspr 517,r4
    blr
FUNC_END(storeNPIDR)
```

And insert a task switch hook within your application

```
void storeNPIDR (WIND_TCB *pOldTcb, WIND_TCB *pNewTcb);
taskSwitchHookAdd((FUNCPTR)storeNPIDR);
```

Function Runtime Statistics

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **Logger**). For details, refer to **"OS-aware Tracing"** (glossary.pdf).

All function-related statistic and time chart evaluations can be used with task-specific information. The function timings will be calculated dependent on the task that called this function. To do this, in addition to the function entries and exits, the task switches must be recorded.

To do a selective recording on task-related function runtimes based on the data accesses, use the following command:

```
; Enable flow trace and accesses to the magic location
Break.Set TASK.CONFIG(magic) /TraceData
```

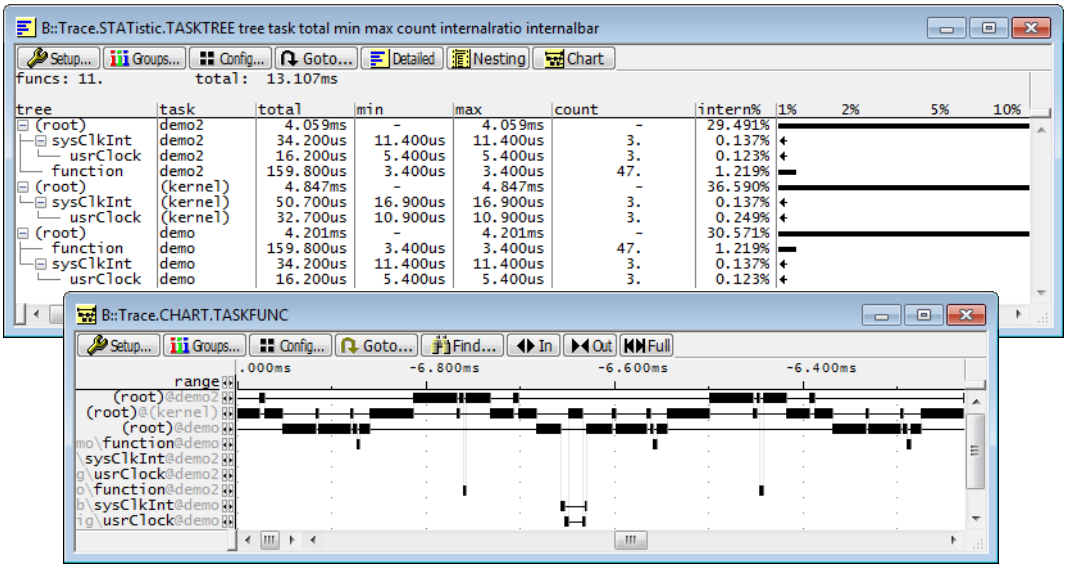
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

To evaluate the contents of the trace buffer, use these commands:

Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat runtime timechart

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".



To correctly detect the function run times, the trace must contain the task switches. See **Task Runtime Statistics** for possibly needed patches.

Task State Analysis

> **NOTE:** This feature is *only* available, if your debug environment is able to trace task switches and data accesses (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate a data trace, or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **Logger**). For details, refer to **"OS-aware Tracing"** (glossary.pdf).

The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically.

This feature requires that the following data accesses are recorded:

- All accesses to the status words of all tasks
- Accesses to the current task variable (= magic address)

Adjust your trace logic to record all data write accesses, or limit the recorded data to the area where all TCBs are located (plus the current task pointer).

Example: This script assumes that the TCBs are located in an array named TCB_array and consequently limits the tracing to data write accesses on the TCBs and the task switch.

```
Break.Set Var.RANGE(TCB_array) /Write /TraceData
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

To evaluate the contents of the trace buffer, use these commands:

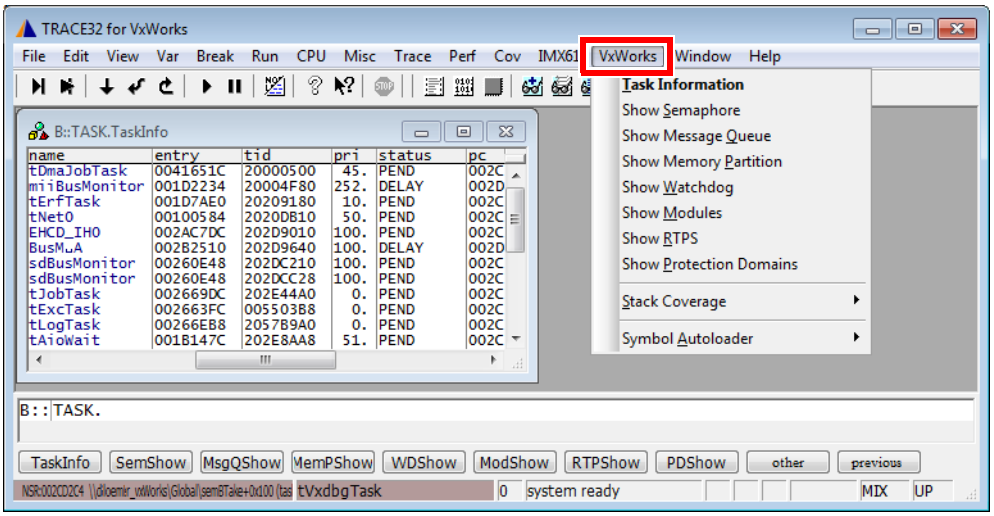
Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state timechart

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".

VxWorks specific Menu

The configuration of the OS Awareness for VxWorks also loads an additional menu with VxWorks specific menu items (see **Configuration**). See the menu file at ~~/demo/*<arch>*/kernel/vxworks/vxworks.men.

You will find a new menu called **VxWorks**.



- **Task Information** opens the **TASK.TaskInfo** window.
- The **Show** menu items launch the appropriate kernel resource display window.
  In VxWorks 5.x you may be asked for the ID to show.
- The **Stack Coverage** submenu starts and resets the VxWorks specific stack coverage and provides an easy way to add or remove tasks from the stack coverage window.
- The **Symbol Autoloader** submenu allows to control the **Symbol Autoloader**.

In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with default display.
- The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime statistics or statistics on task states.

Right-clicking a variable shows an additional **VxWorks** submenu that allows to show this variable as a specific VxWorks object.

TASK.LKUP — Show system symbol table

Format: **TASK.LKUP** [**0**|*<address>*|*<symbol>* [**0**|*<module_id>*|*<module_name>*] [*<section_name>*]]]

Displays the target symbol table with the specified filtering.

Examples:

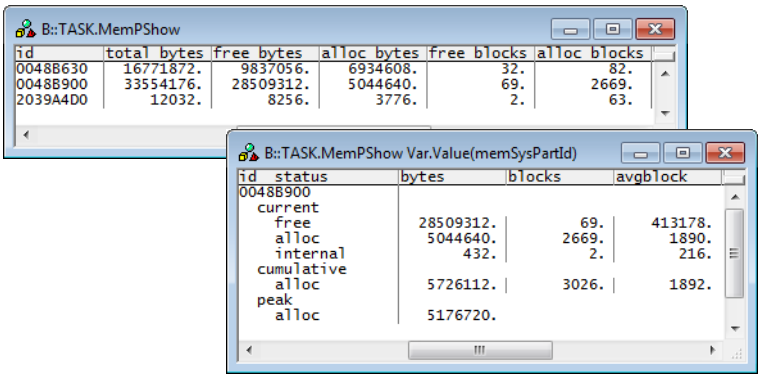
```
TASK.LKUP                          ;shows all symbols

TASK.LKUP "mysymbol"               ;shows only the entry of "mysymbol"

TASK.LKUP 0 "apps.out"             ;shows all symbols of module "apps.out"
                                   ;(null is the specifier for all)

TASK.LKUP 0 0 "common"             ;shows all symbols of the common section

TASK.LKUP 0 "apps.out" "common"    ;shows all symbols of the common section
                                   ;of apps.out.
```

TASK.MemPShow — Show memory partition

Format: **TASK.MemPShow** *<memory_partition>*

Displays the memory partition table of VxWorks or detailed information about one specific memory partition.

Without any arguments, a table with all created memory partition will be shown.
Specify a memory partition ID to display detailed information about this memory partition.



TASK.MMU.SCAN Scan RTP MMU entries

Format:	**TASK.MMU.SCAN** [*<rtp_id>* [*<address>*] [*<size>*]]] (deprecated) **Use full MMU Support instead.**

Scan the MMU entries of RTPs.

This command copies the address translation of an RTP into the debugger's MMU. See **TRANSlation.List**.

Without any argument, the MMU translation tables of all RTPs are scanned.

The first parameter specifies the RTP ID to scan.

Optionally this command takes a start address and a size parameter to restrict the scanned address range. The size defaults to 0x08000000, if only an address is given.

TASK.MMU.SCANSPACE Scan PD MMU entries

Format:	**TASK.MMU.SCANSPACE** [*<pdid>* [*<address>*] [*<size>*]]] (deprecated) **Use full MMU Support instead.**

Scan the MMU entries of protection domains.

This command copies the address translation of a PD into the debugger's MMU. See **TRANSlation.List**.

Without any argument, the MMU translation tables of all PDs are scanned.

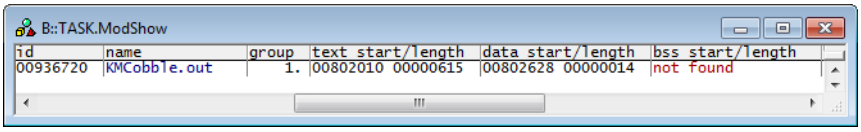
The first parameter specifies the space ID (= partition ID) to scan. Check **TASK.PDShow** for the space ID.

Optionally this command takes a start address and a size parameter to restrict the scanned address range. The size defaults to 0x08000000, if only an address is given.

TASK.ModShow — Show loaded modules

Format:	**TASK.ModShow**

Displays a table with all loaded modules.

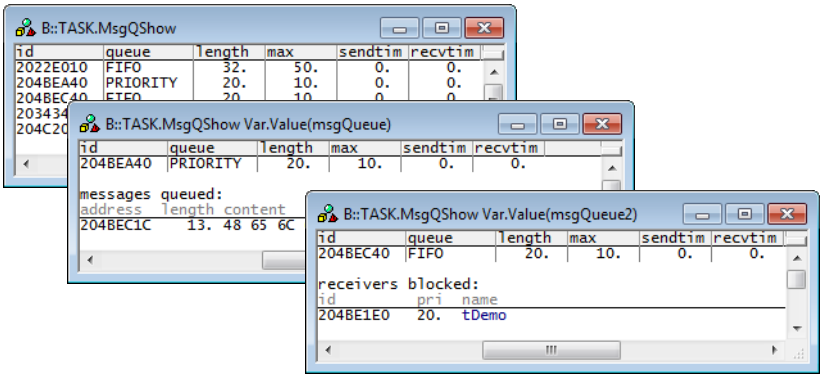


TASK.MsgQShow — Show message queues

Format:	**TASK.MsgQShow** *<msg_queue>*

Displays the message queue table of VxWorks or detailed information about one specific message queue.

Without any arguments, a table with all created message queues will be shown.
Specify a message queue ID to display detailed information about this message queue.



TASK.Option — Set awareness options

Format:	**TASK.Option** *<option>*
<option>:	**THRCTX** [**ON** | **OFF**]

Set various options to the awareness.

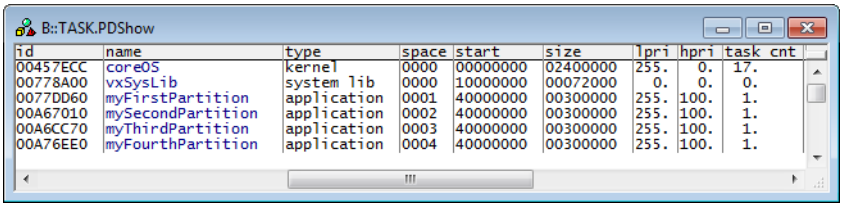
THRCTX Set the context ID type that is recorded with the real-time trace (e.g. ETM). If set to on, the context ID in the trace contains thread switch detection. See **Task Runtime Statistics**.

TASK.PDShow — Show protection domains

Format:	**TASK.PDShow** [*<pd_id>*]

Displays a table with all created protection domains (aka ARINC653 partitions) or detailed information about one specific PD.

Without any arguments, a table with all protection domains will be shown.
Specify a PD ID to display detailed information on that PD.



B::TASK.PDShow

id	name	type	space	start	size	lpri	hpri	task cnt
00457ECC	coreOS	kernel	0000	00000000	02400000	255.	0.	17.
00778A00	vxSysLib	system lib	0000	10000000	00072000	0.	0.	0.
0077DD60	myFirstPartition	application	0001	40000000	00300000	255.	100.	1.
00A67010	mySecondPartition	application	0002	40000000	00300000	255.	100.	1.
00A6CC70	myThirdPartition	application	0003	40000000	00300000	255.	100.	1.
00A76EE0	myFourthPartition	application	0004	40000000	00300000	255.	100.	1.

TASK.RELOC Relocate system symbols

Format: **TASK.RELOC** [**0**|*<address>*|*<symbol>* [**0**|*<module_id>*|*<module_name>*] [*<section_name>*]]]

This command takes the same parameters as **TASK.LKUP** and relocates the symbols to their correct addresses.

Example:

```
TASK.RELOC 0 "apps.out" "common"
```

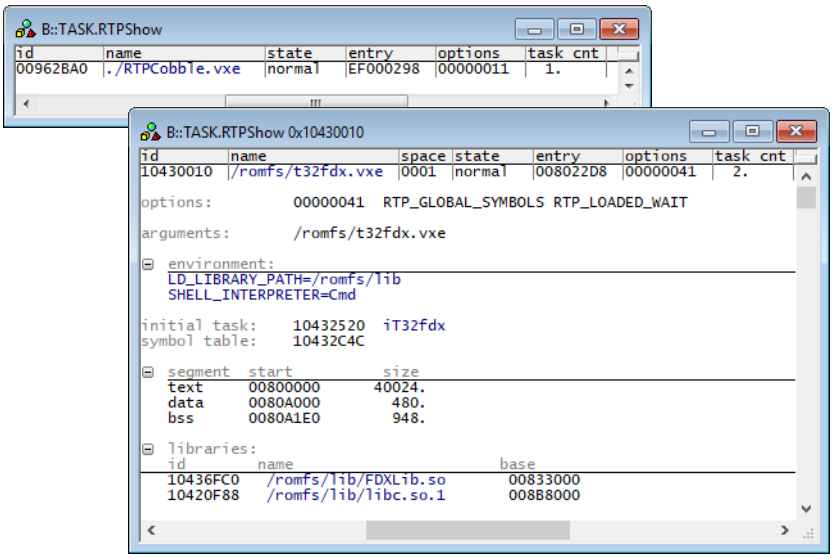
Relocates all symbols of the COMMON section of the apps.out module.

TASK.RTPShow Show loaded RTPs

Format: **TASK.RTPShow** [*<rtp>*]

Displays a table with all loaded RTPs or detailed information about one specific RTP.

Without any arguments, a table with all loaded RTPs will be shown.
Specify an RTP name or ID to display detailed information on that RTP.

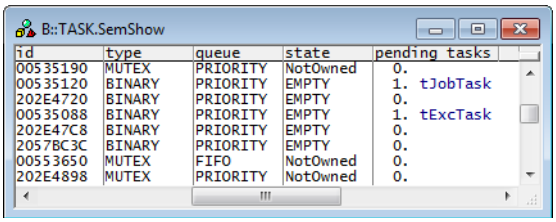


TASK.SemShow — Show semaphores

Format: **TASK.SemShow** *<semaphore>*

Displays the semaphore table of VxWorks or detailed information about one specific semaphore.

Without any arguments, a table with all created semaphores will be shown.
Specify a semaphore ID to display detailed information about this semaphore.



id	type	queue	state	pending tasks
00535190	MUTEX	PRIORITY	NotOwned	0.
00535120	BINARY	PRIORITY	EMPTY	1. tJobTask
202E4720	BINARY	PRIORITY	EMPTY	0.
00535088	BINARY	PRIORITY	EMPTY	1. tExcTask
202E47C8	BINARY	PRIORITY	EMPTY	0.
2057BC3C	BINARY	PRIORITY	EMPTY	0.
00553650	MUTEX	FIFO	NotOwned	0.
202E4898	MUTEX	PRIORITY	NotOwned	0.

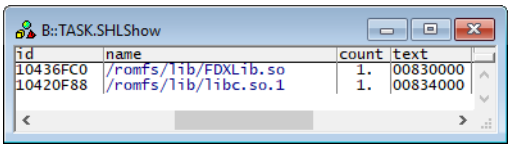
The column “pending tasks” contains the number of tasks pending in the first place, following the task names.

TASK.SHLShow — Show loaded libraries

Format: **TASK.SHLShow** [*<library>*]

Displays a table with all loaded libraries or detailed information about one specific library.

Without any arguments, a table with all loaded libraries will be shown.
Specify a library name or ID to display detailed information on that library.



id	name	count	text
10436FC0	/romfs/lib/FDXLib.so	1.	00830000
10420F88	/romfs/lib/libc.so.1	1.	00834000

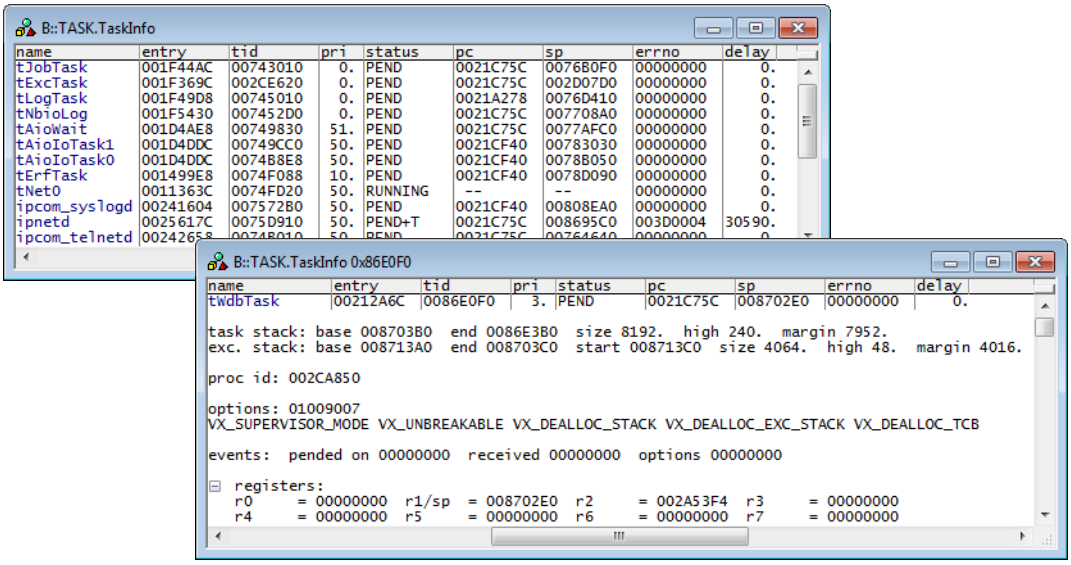
TASK.TaskInfo — Task information

Format: **TASK.TaskInfo** *<task>*

Displays the task table of VxWorks or detailed information about one specific task.

The display is similar to the “i” command of the VxWorks shell.

Without any arguments, a table with all created tasks will be shown.
Specify a task name or task magic number to display detailed information on that task.



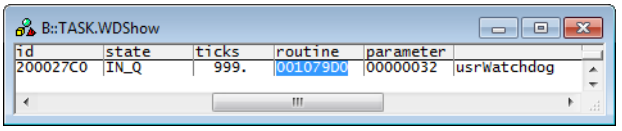
The task ID ('tid') is equal to the “magic number” of this task.
The “pc” and “sp” columns show the program counter resp. the stack pointer of the task on the stack (only available if task is not running).
The fields “name”, “entry”, “tid” and “pc” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on the “tid” will show a local menu.

TASK.WDShow — Show watchdogs

Format:	**TASK.WDShow** <*watchdog*>

Displays the watchdog table of VxWorks or detailed information about one specific watchdog.

Without any arguments, a table with all created watchdogs will be shown.
Specify a watchdog ID to display detailed information about this watchdog.



The “routine” field is mouse sensitive.

VxWorks PRACTICE Functions

There are special definitions for VxWorks specific PRACTICE functions.

TASK.AVAIL() Availability of object lists

Syntax:	**TASK.AVAIL(***<item>***)**
<item>:	**semlist** | **msgqlist** | **memplist** | **wdlist**

Reports availability of object lists.

Parameter Type: String (***without*** quotation marks).

Return Value Type: Hex value.

TASK.CONFIG() OS Awareness configuration information

Syntax:	**TASK.CONFIG(magic** | **magic:***<core>* | **magicsize)**

Parameter and Description:

magic	**Parameter Type**: String (***without*** quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number).
magic:*<core>*	**Parameter Type**: String (***without*** quotation marks). Returns the address for the magic number of the given core ID.
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.

TASK.MODLIST() Next module magic number

Syntax:	**TASK.MODLIST(***<module_magic>***)**

Returns the next module magic number in the module list.

Parameter Type: Decimal or hex or binary value. Specify zero for the first module.

Return Value Type: Hex value. Returns zero if no further module available.

TASK.MODNAME() Module name of module

Syntax:	**TASK.MODNAME(***<module_magic>***)**

Returns the module name for the specified module magic number.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.MODULE() Segment address of module

Syntax:	**TASK.MODULE("***<module_name>***",***<segment_id>***)**
<segment_id>:	**0** | **1** | **2** | **3**

Reports the segment address of a given module.

Parameter and Description:

<module_name>	**Parameter Type**: String (*with* quotation marks).
0, **1**, **2**, **3**	**Parameter Type**: Decimal or hex or binary value. **0**=text, **1**=data, **2**=bss, **3**=common

Return Value Type: Hex value.

TASK.RTP.ID() RTP ID of rtp name

Syntax: **TASK.RTP.ID("*<rtp_name>*")**

Returns the RTP ID of a given rtp name.

Parameter Type: String (*with* quotation marks).

Return Value Type: Hex value.

TASK.RTP.SEGADDR() Segment address of RTP

Syntax: **TASK.RTP.SEGADDR("*<segment_name>*",*<rtp_id>*)**

Returns the segment address of a given segment name and RTP ID.

Parameter and Description:

<segment_name>	**Parameter Type**: String (*with* quotation marks).
<rtp_id>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.RTP.SEGSIZE() Segment size of RTP

Syntax: **TASK.RTP.SEGSIZE("*<segment_name>*",*<rtp_id>*)**

Returns the segment size of a given segment name and RTP ID.

Parameter and Description:

<segment_name>	**Parameter Type**: String (*with* quotation marks).
<rtp_id>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.RTP.SPACEID() Space ID of RTP ID

Syntax: **TASK.RTP.SPACEID(**<*rtp_id*>**)**

Returns the space ID of a given RTP ID.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.RTP.TTB() TTB address of RTP ID

Syntax: **TASK.RTP.TTB(**<*rtp_id*>**)**

Returns the TTB address of a given RTP ID.

Parameter Type: Decimal or hex or binary value. Specify zero to get the TTB of the kernel.

Return Value Type: Hex value.

TASK.SHL.ID() ID of library name

Syntax: **TASK.SHL.ID("**<*shl_name*>**")**

Returns the ID of a given library name.

Parameter Type: String (*with* quotation marks).

Return Value Type: Hex value.

TASK.SHL.SEGADDR() Segment address of library

Syntax: **TASK.SHL.SEGADDR("***<segment_name>***",***<shl_id>***)**

Returns the segment address of a given segment name and library ID.

Parameter and Description:

<segment_name>	**Parameter Type**: String (*with* quotation marks).
<shl_id>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.SHL.SEGSIZE() Segment size of library

Syntax: **TASK.SHL.SEGSIZE("***<segment_name>***",***<shl_id>***)**

Returns the segment size of a given segment name and library ID.

Parameter and Description:

<segment_name>	**Parameter Type**: String (*with* quotation marks).
<shl_id>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.TASKLIST() Next task magic number in task list

Syntax: **TASK.TASKLIST(***<task_magic>***)**

Returns the next task magic number in the task list.

Parameter Type: Decimal or hex or binary value. Specify zero for the first task.

Return Value Type: Hex value. Returns zero if no further task available.

Syntax: **TASK.TASKNAME(**<*task_magic*>**)**

Returns the task name of the specified task.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.