OS Awareness Manual PXROS

OS Awareness Manual PXROS

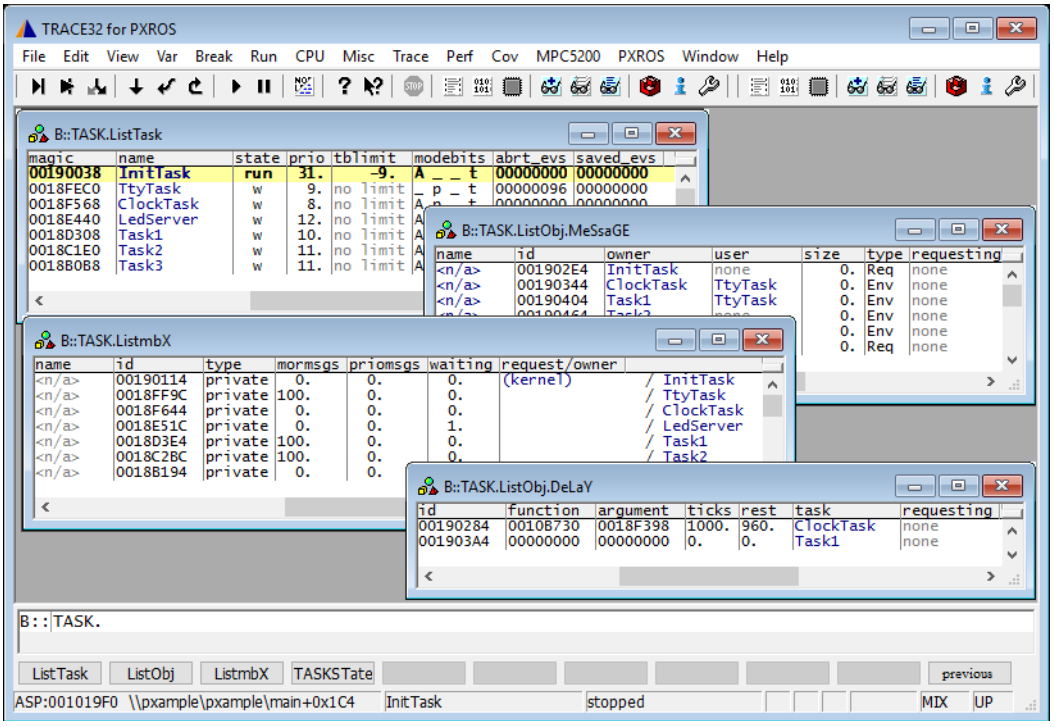
Version 18-Nov-2020

History

08-Oct-19 Added support for PXROS v7

28-Aug-18 The title of the manual was changed from “RTOS Debugger for *PXROS*” to “OS Awareness Manual *PXROS*”.

Overview



The OS Awareness for PXROS 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:

- **"Debugger Basics - Training"** (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 PXROS is supported for the following versions:

- PXROS 4.x on C166/C167, PowerPC and TriCore
- PXROS 5.x, 6.x and 7.x on TriCore

Configuration

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

TASK.CONFIG ~~/demo/*<arch>***/kernel/pxros/pxros.t32** [*<magic_address>* [*<args>*]]

<magic_address> Specifies a memory location that contains the current running task. This address can be found at “. . .”.

<args> The configuration requires additional arguments, that are:

- *<sleep>*: Currently not used, specify “0”
- *<dpp>:* (only on C166) The first argument configures the data page settings of the application. Specify a long word which least significant byte is the dpp0 content and which most significant byte is the dpp3 content. E.g. '03060500' means dpp0=0, dpp1=5, dpp2=6 and dpp3=3. If you don't know the dpp settings of your application, just start it for a while and check in the 'register' command the dpp's. Note that the dpp settings must be adapted to every single application.
- *<internal>*: The next three arguments are PXROS internal structures. Specify “`__PxTasklist __PxTaskRdyFromRdy __PxUsedObjs`”.

Without any parameters, the debugger tries to locate the internals of PXROS automatically. For this purpose, the kernel symbols must be loaded and accessible at any time the OS Awareness is used (see also “**Hooks & Internals**”).

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). In case of a ICE or FIRE, you have to map emulation or shadow memory to the address space of all used system tables.

Quick Configuration Guide

Example scripts are provided in ~~/demo/*<arch>*/kernel/pxros. It is recommended to take one of these as a starting point and modify it to suit your target and setup.

If you already have a setup/configuration script which configures the target and loads the application code and/or symbols, you can add the following lines to your script after the symbols have been loaded:

```
TASK.CONFIG ~~/demo/<arch>/kernel/pxros/pxros.t32
MENU.ReProgram ~~/demo/<arch>/kernel/pxros/pxros.men
```

These lines will automatically configure the awareness and add a custom menu that provides access to many of the features.

Hooks & Internals in PXROS

No hooks are used in the kernel.

To retrieve information on 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.

Debug Features

The OS Awareness for PXROS supports the following debug features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. Information on the following PXROS components can be displayed:

TASK.ListObject.DeLaY	Delay objects
TASK.ListObject.MailBoX or **TASK.ListmbX**	Mailboxes
TASK.ListObject.MemClass	Memory classes
TASK.ListObject.MeSsaGe	Message objects
TASK.ListObject.OPool	Object pools
TASK.ListTask	Tasks

For a description of the commands, refer to chapter "**PXROS 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.

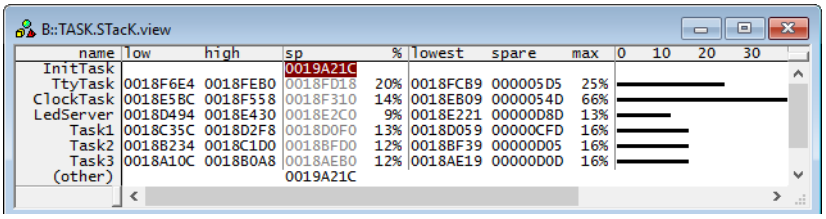
Task Stack Coverage

For stack usage coverage of the 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, flag memory must be mapped to the task stack areas when working with the emulation memory. When working with the target memory, 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.



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.

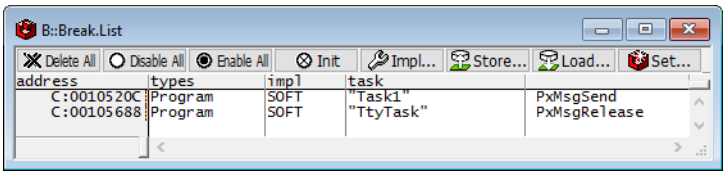
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

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.

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.

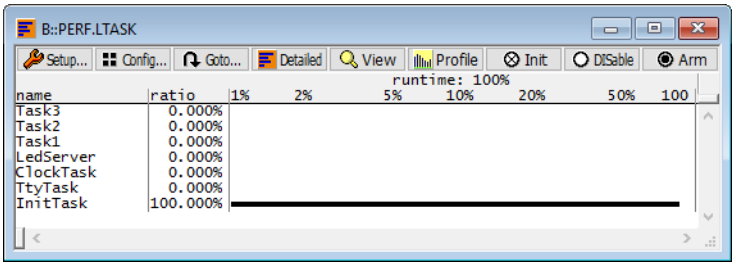
Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

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.

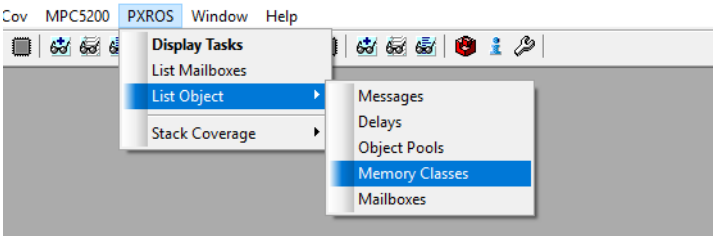
For a general description of the **PERF** command group, refer to **“General Commands Reference Guide P”** (general_ref_p.pdf).



PXROS Specific Menu

The menu file “pxros.men” contains a menu with PXROS specific menu items. Load this menu with the **MENU.ReProgram** command.

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



- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the PXROS specific stack coverage and provides an easy way to add or remove tasks from the stack coverage window.

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

- The **Trace**, **List** menu is extended.
  - “Task Switches” shows a trace list window with only task switches (if any)
  - “Default and Tasks” shows switches together with the default display.
- The **Perf** menu contains additional submenus
  - “Task Runtime” enables and shows the **task runtime analysis**
  - “Task Function Runtime” enables and shows the **function runtime statistics** based on tasks
  - “CPU Load” enables and shows the **CPU load analysis**

Trace Features

The OS Awareness for PXROS supports the following trace features.

Task Runtime Statistics

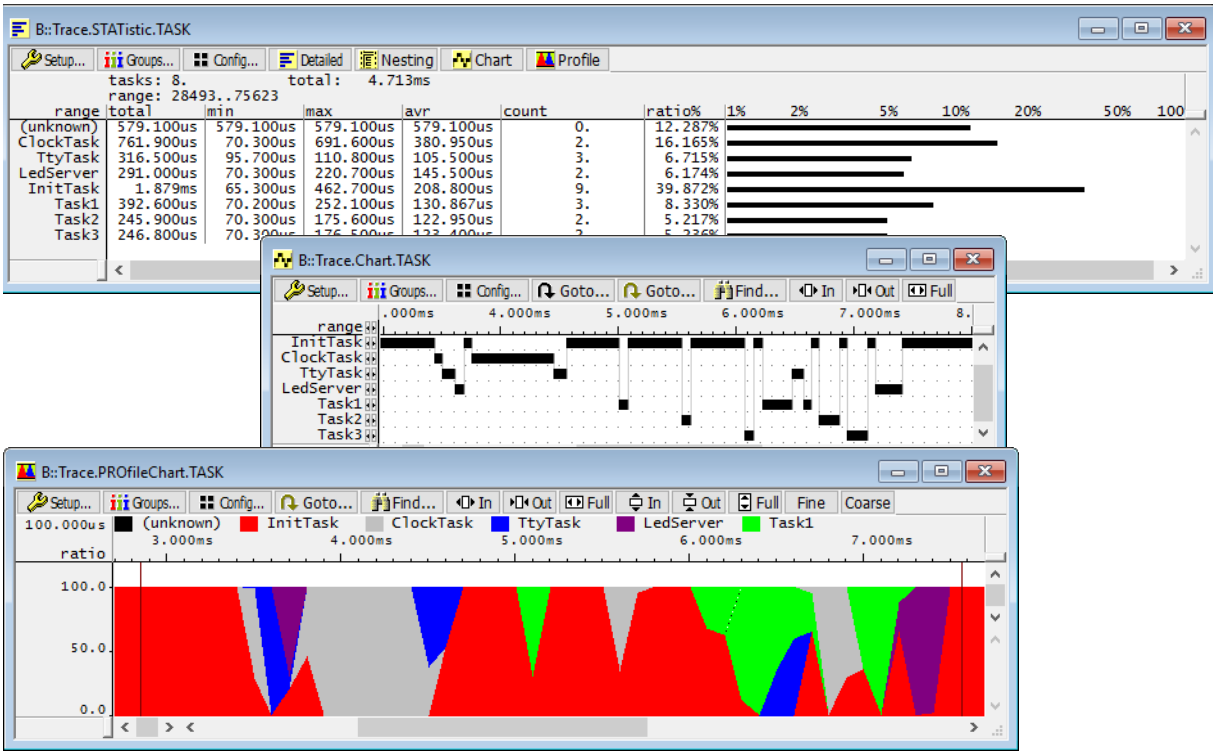
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software 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)".



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
```

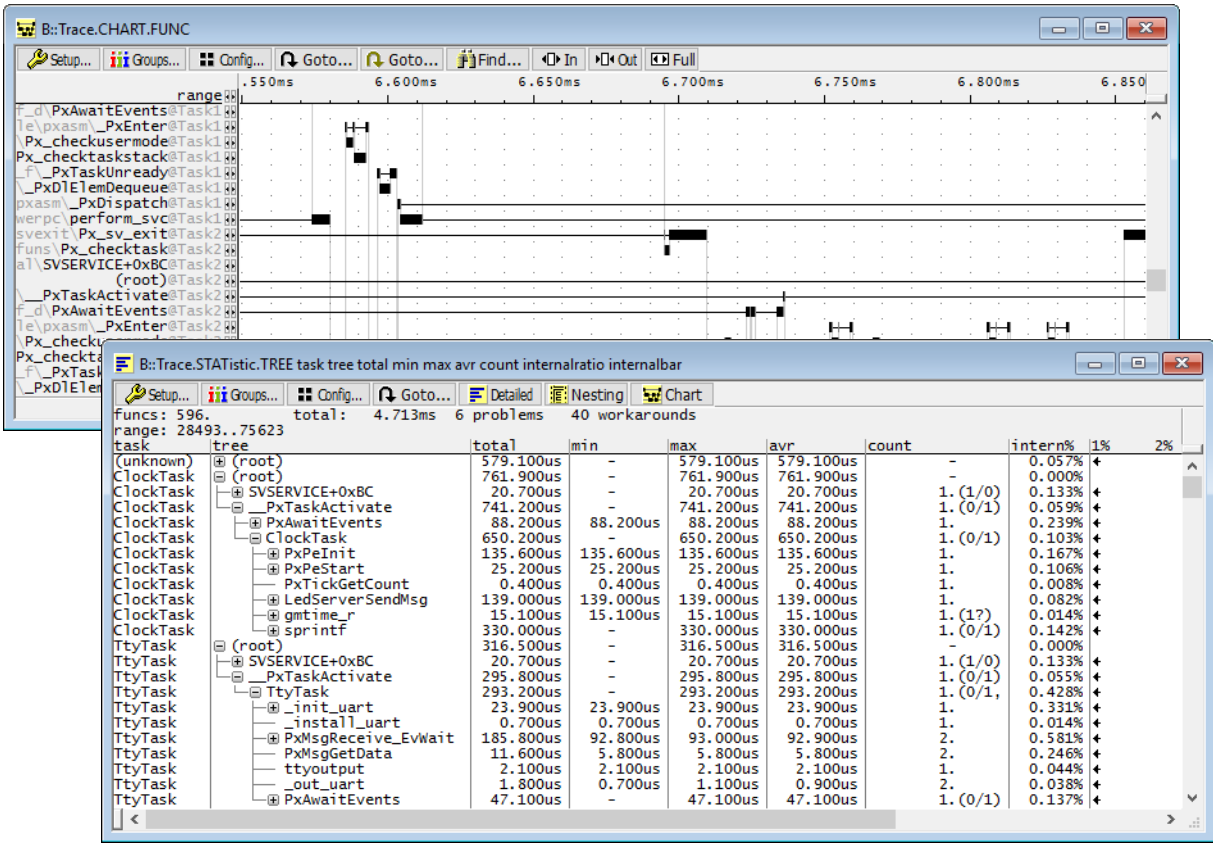
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



task	tree	total	min	max	avr	count	intern%	1% 2%
(unknown)	⊞ (root)	579.100us	-	579.100us	579.100us	-	0.057%	
ClockTask	⊟ (root)	761.900us	-	761.900us	761.900us	-	0.000%	
ClockTask	⊞ SVSERVICE+0xBC	20.700us	-	20.700us	20.700us	1.(1/0)	0.133%	
ClockTask	⊟ __PxTaskActivate	741.200us	-	741.200us	741.200us	1.(0/1)	0.059%	
ClockTask	⊞ PxAwaitEvents	88.200us	88.200us	88.200us	88.200us	1.	0.239%	
ClockTask	⊟ ClockTask	650.200us	-	650.200us	650.200us	1.(0/1)	0.103%	
ClockTask	⊞ PxPeInit	135.600us	135.600us	135.600us	135.600us	1.	0.167%	
ClockTask	⊞ PxPeStart	25.200us	25.200us	25.200us	25.200us	1.	0.106%	
ClockTask	PxTickGetCount	0.400us	0.400us	0.400us	0.400us	1.	0.008%	
ClockTask	⊞ LedServerSendMsg	139.000us	139.000us	139.000us	139.000us	1.	0.082%	
ClockTask	⊞ gmtime_r	15.100us	15.100us	15.100us	15.100us	1.(1?)	0.014%	
ClockTask	⊞ sprintf	330.000us	-	330.000us	330.000us	1.(0/1)	0.142%	
TtyTask	⊟ (root)	316.500us	-	316.500us	316.500us	-	0.000%	
TtyTask	⊞ SVSERVICE+0xBC	20.700us	-	20.700us	20.700us	1.(1/0)	0.133%	
TtyTask	⊟ __PxTaskActivate	295.800us	-	295.800us	295.800us	1.(0/1)	0.055%	
TtyTask	⊟ TtyTask	293.200us	-	293.200us	293.200us	1.(0/1,	0.428%	
TtyTask	⊞ _init_uart	23.900us	23.900us	23.900us	23.900us	1.	0.331%	
TtyTask	_install_uart	0.700us	0.700us	0.700us	0.700us	1.	0.014%	
TtyTask	⊞ PxMsgReceive_EvWait	185.800us	92.800us	93.000us	92.900us	2.	0.581%	
TtyTask	PxMsgGetData	11.600us	5.800us	5.800us	5.800us	2.	0.246%	
TtyTask	ttyoutput	2.100us	2.100us	2.100us	2.100us	1.	0.044%	
TtyTask	_out_uart	1.800us	0.700us	1.100us	0.900us	2.	0.038%	
TtyTask	⊞ PxAwaitEvents	47.100us	-	47.100us	47.100us	1.(0/1)	0.137%	

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 CPU load. The CPU load is calculated by comparing the time spent in all tasks against the time spent in the idle task. The measurement is done by using the **GROUP** command to group all idle tasks and calculating the time spent in all other tasks.

Example: Two idle tasks named "IdleTask1" and "IdleTask2":

```
; Create a group called "idle" with the idle tasks
GROUP.CreateTASK "idle" "IdleTask1"
GROUP.CreateTASK "idle" "IdleTask2"

; Unmark "idle" and mark all others in red
GROUP.COLOR "idle"  NONE
GROUP.COLOR "other" RED

; Merge idle tasks and other tasks
GROUP.MERGE "idle"
GROUP.MERGE "other"
```

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

Trace.STATistic.TASK	Display CPU load statistic evaluation
Trace.PROfileChart.TASK	Display CPU load as colored graph

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

When CPU load analysis is no longer needed, or if a detailed **Task Runtime Statistic** is needed, disable the grouping of the tasks with:

```
;comments
GROUP.SEParate "idle"
GROUP.SEParate "other"
```

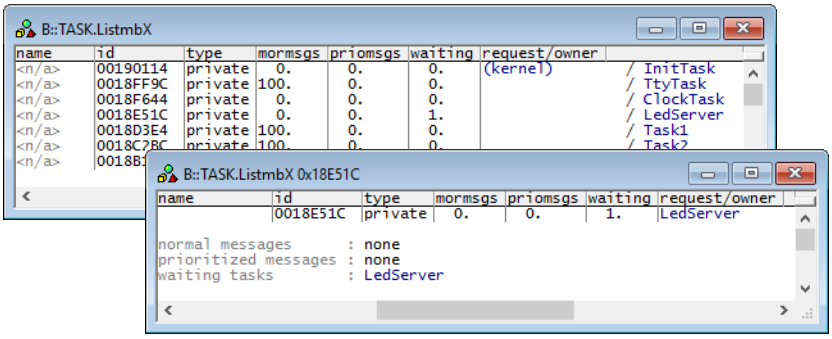
PXROS Specific Menu for Tracing

The menu entries specific to tracing are already described in the **menu for debug features**.

TASK.ListmbX — Display mailboxes

Format: **TASK.ListmbX** <*mbx_id*>

This command is just an alias for **Task.ListObj.MailBoX**. See there for a description.



TASK.ListObject — List objects

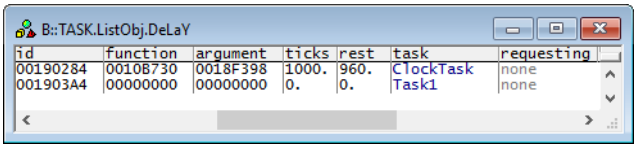
Format: **TASK.ListObject.**[<*object*>]

<*object*>: **MeSsaGe** | **DeLaY** | **OPool** | **MemClass** | **MailBoX**

List PXROS objects. See detailed descriptions below.

Format: **TASK.ListObj.DeLaY**

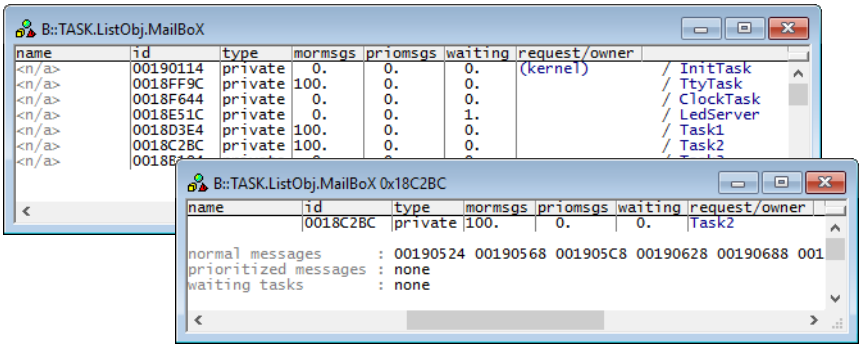
Displays a table of the delay objects in the system.



TASK.ListObj.MailBoX Display mailboxes

Format: **TASK.ListObj.MailBoX** <*mbx_id*>

Without any argument, this command displays all system and private mailboxes.With a mailbox a mailbox id as an argument, it shows the specified mailbox with it's pending messages and waiting tasks.

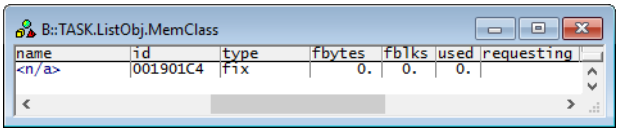


Format: **TASK.ListObj.MemClass**

Displays a table of the memory classes.

The 'type' field contains the memory class type. If this is fixed, the 'blksize' field contains the block size.

'fbytes' and 'fblks' contain the free bytes and free blocks in that mc.



TASK.ListObj.MeSsaGe Display message objects

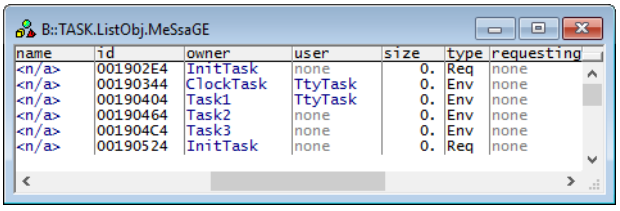
Format: **TASK.ListObj.MeSsaGe**

Displays a table of the message objects in the system.

The 'data' field shows the pointer to the message data.

The 'size' field specifies the message size, while 'buff' is the siye of the entire data area.

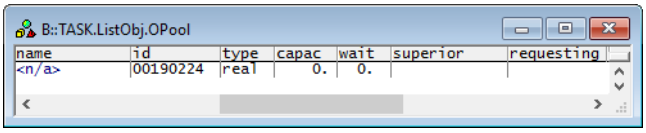
The 'type' is either 'Req' for 'PxMsgRequest' or Env for 'PxMsgEnvelop'.



Format: **TASK.ListObj.Opool**

Displays a table of the object pools.

The 'wait' column contains the number of waiting tasks.

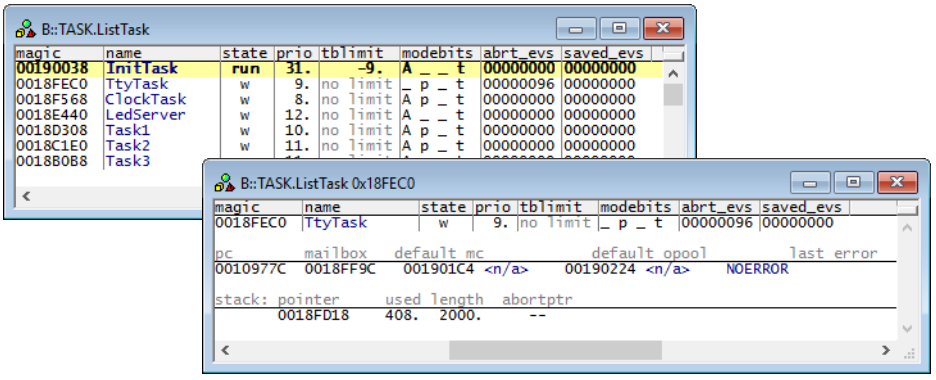


TASK.ListTask Display task table

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

Without any argument this command displays a list of tasks. For an explanation of the mode bits check the PXmon manual.

With an ID or a task name as an argument, you get a detailed description of that task.



Frequently-Asked Questions

No information available