

OS Awareness Manual SMX

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

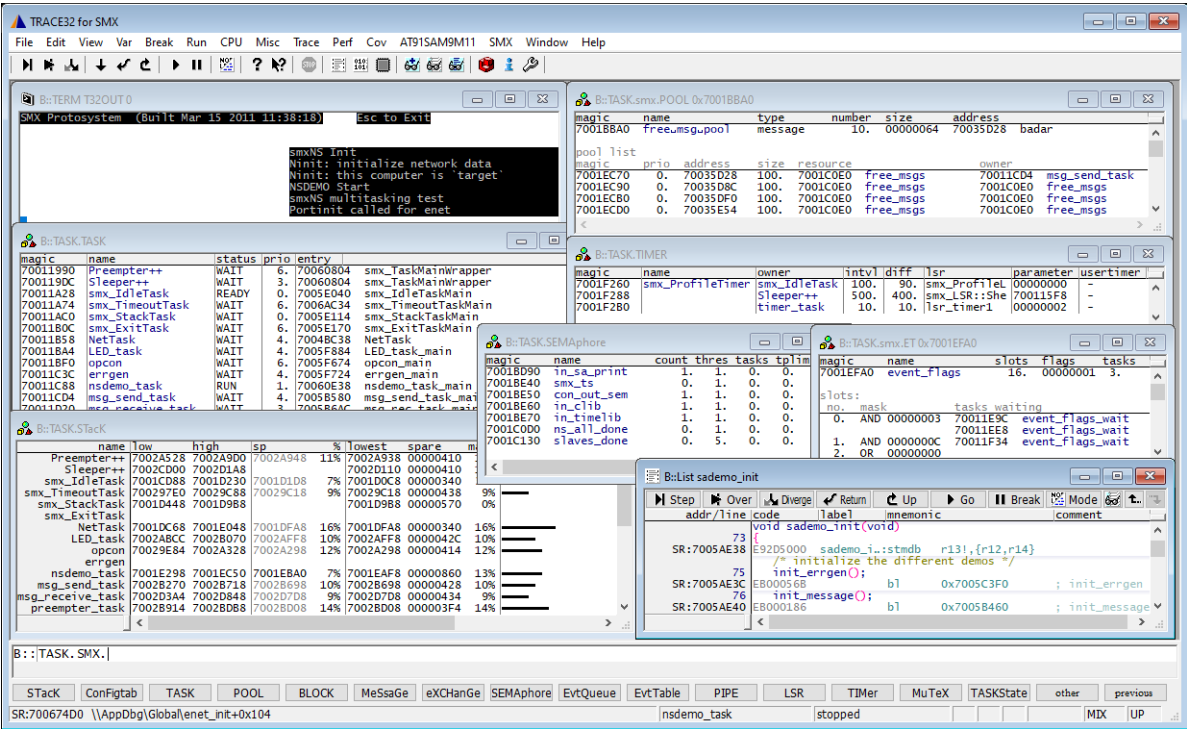
TRACE32 Documents	
OS Awareness Manuals	
OS Awareness Manual SMX	1
History	4
Overview	4
Brief Overview of Documents for New Users	5
Supported Versions	5
Configuration	6
Quick Configuration Guide	7
Hooks & Internals in SMX	7
Features	8
Display of Kernel Resources	8
Task Stack Coverage	8
Task-Related Breakpoints	9
Task Context Display	10
Dynamic Task Performance Measurement	11
Task Runtime Statistics	12
Task State Analysis	13
Function Runtime Statistics	14
SMX specific Menu	16
SMX Commands	17
TASK.BLOCK	Display blocks 17
TASK.BUCKet	Display buckets 17
TASK.ConFigtab	Display configuration 18
TASK.EvtQueue	Display event queues 18
TASK.EvtTable	Display event tables 19
TASK.eXCHanGe	Display exchanges 20
TASK.LSR	Display LSRs 20
TASK.MeSsaGe	Display messages 21
TASK.PIPE	Display pipes 21
TASK.POOL	Display pools 22
TASK.SEMaphore	Display semaphores 23
TASK.TASK	Display tasks 24

TASK.TIMER	Display timers	24
TASK.TRACE	Display event buffer	25
TASK.TRACEVM	Copy event buffer to LOGGER	25
SMX PRACTICE Functions		27
TASK.CONFIG()	OS Awareness configuration information	27

History

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

Overview



The OS Awareness for SMX 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 the SMX awareness is tested on the following versions:

- SMX V3.4 to V4.0 on ARM, PowerPC and SH.

Configuration

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

Automatic configuration tries to locate the SMX internals automatically. For this purpose all symbol tables must be loaded and accessible at any time the OS Awareness is used.

If you want to have dual port access for the display functions (display “On The Fly”), you have to map emulation memory to the address space of all used system tables.

For system resource display and analyzer functionality, you can do an automatic configuration of the OS Awareness. For this purpose it is necessary that all system internal symbols are loaded and accessible. Each of the **TASK.CONFIG** arguments can be substituted by '0', which means that this argument will be searched and configured automatically. For a fully automatic configuration, omit all arguments:

Format: TASK.CONFIG smx

Quick Configuration Guide

To access all features of the OS Awareness you should follow the following roadmap:

1. Run the PRACTICE demo script (`~/demo/<processor>/kernel/smx/smx.cmm`). Start the demo with `"do smx"` and `"go"`. The result should be a list of tasks, which continuously change their state.
2. Make a copy of the PRACTICE script file `"smx.cmm"`. Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the kernel resources and use the analyzer functions.

Hooks & Internals in SMX

No hooks are used in the kernel.

To retrieve information on kernel objects, the OS Awareness uses the global SMX variables and structures exported by the SMX library, and the structures defined in the `smx.h` file. Be sure that your application is compiled and linked with debugging symbols switched on. The SMX library may be compiled without debugging symbols.

SMX provides a mechanism for debugging called "Handle Table". TRACE32 does not use this handle table for SMX aware debugging. The handle table is only used for the resource names (exception: event table overview). If you omit the handle table from your application, you will just lose the display of the resource names.

Features

The OS Awareness for SMX supports the following features.

Display of Kernel Resources

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

TASK.BLOCK	Block
TASK.BUCKet	Buckets
TASK.ConFigtAb	Configuration
TASK.EvtQueue	Event Queues
TASK.EvtTable	Event Tables
TASK.eXCHanGe	Exchanges
TASK.LSR	LSRs
TASK.MeSsaGe	Messages
TASK.PIPE	Pipes
TASK.POOL	Pools
TASK.SEMaphore	Semaphores
TASK.TASK	Tasks
TASK.TIMer	Timers

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

When working with emulation memory or shadow memory, these resources can be displayed “On The Fly”, i.e. while the target application is running, without any intrusion to the application. If using this dual port memory feature, be sure that emulation memory is mapped to all places, where SMX holds its tables.

When working only with target memory, the information will only be displayed if the target application is stopped.

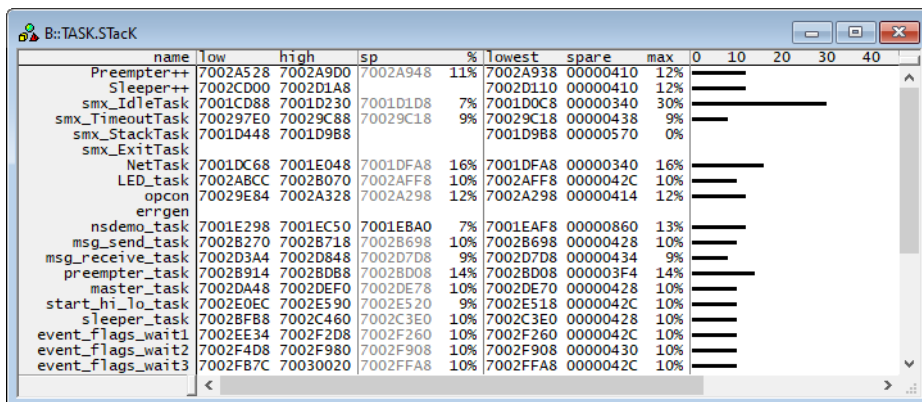
Task Stack Coverage

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.



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.

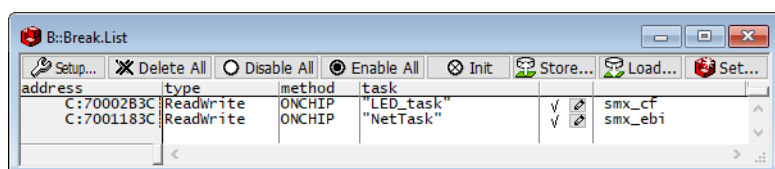
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

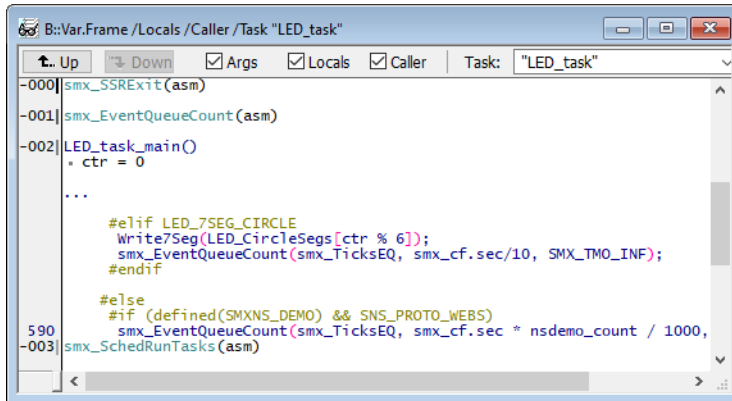
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.

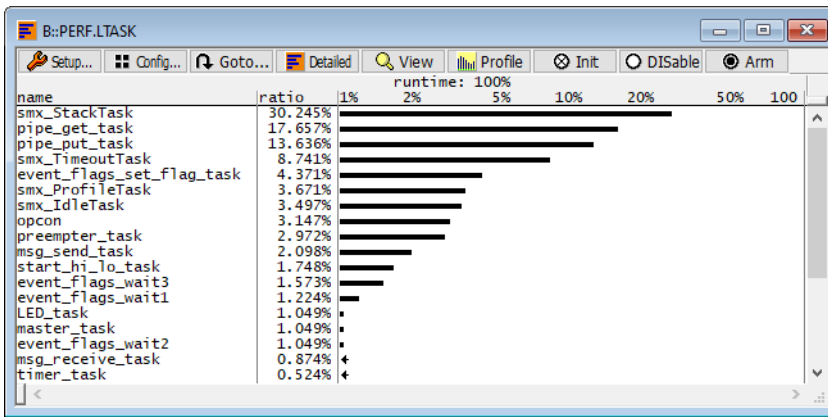


Dynamic Task Performance Measurement

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:

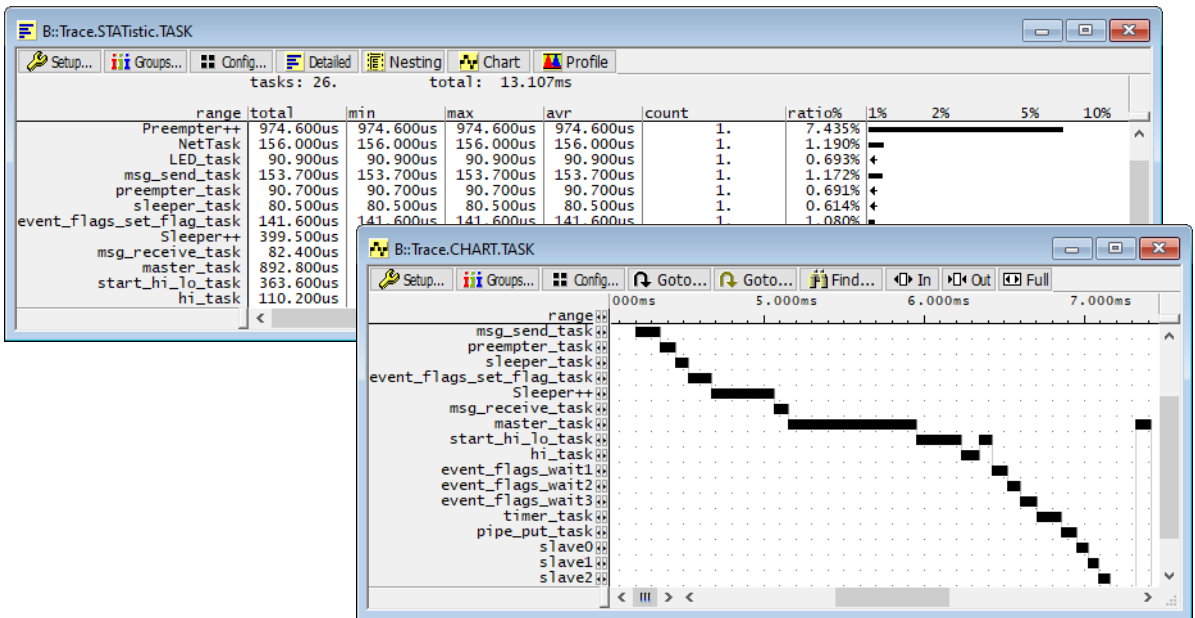
This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based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

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
```

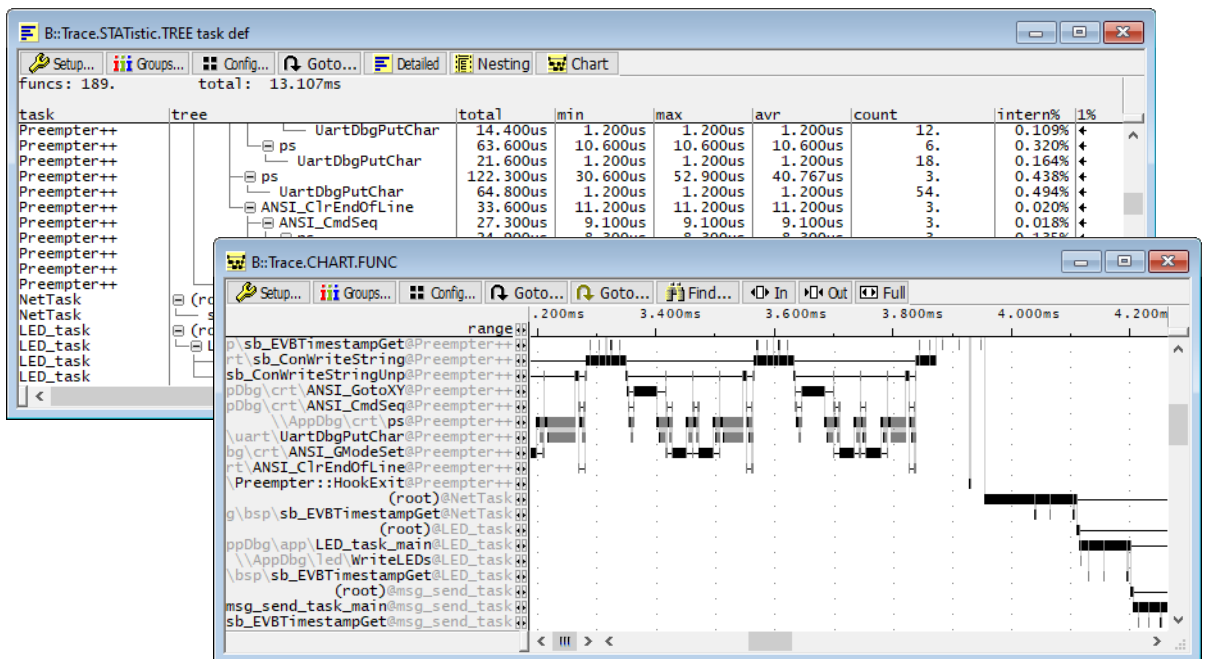
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



SMX specific Menu

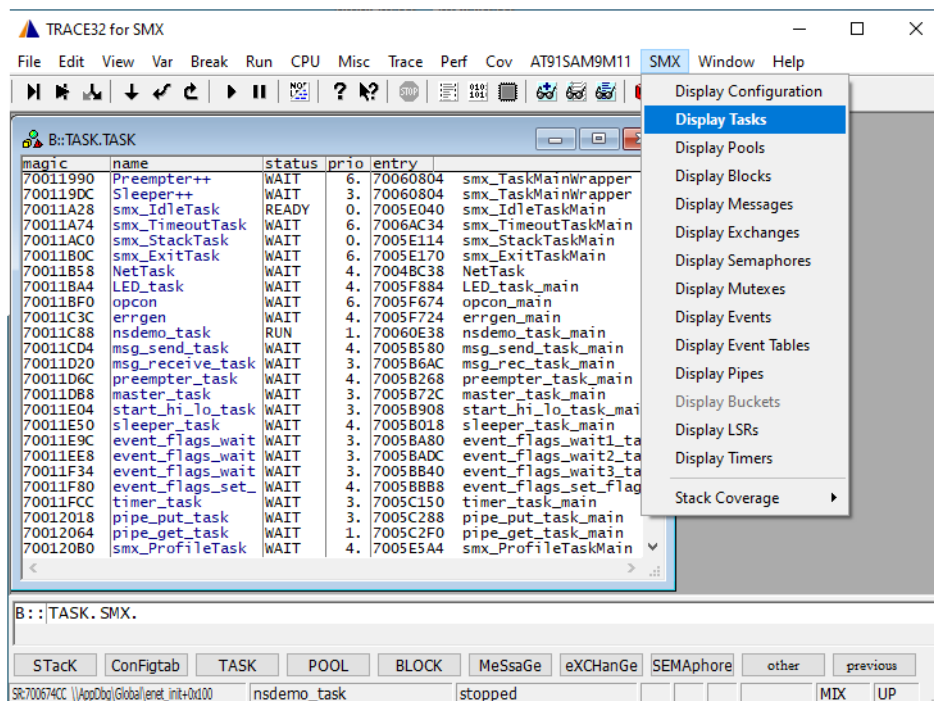
The menu file “smx.men” contains a menu with SMX specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- The **Display** menu items launch the appropriate kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the SMX specific stack coverage, and provide 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** submenu is extended. You can additionally choose if you want a trace list window to show only task switches (if any) or task switches and defaults.
- The **Perf** menu contains the additional submenus for task runtime statistics amd task-related function runtime statistics. For the function runtime statistics, a PRACTICE script file called “men_ptfp.cmm” is used. This script file must be adapted to your application.



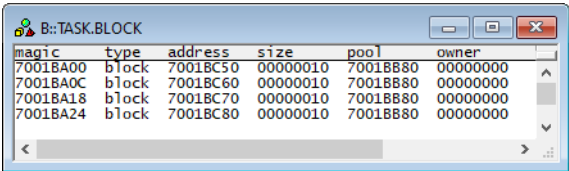
TASK.BLOCK

Display blocks

Format: **TASK.BLOCK** <block>

Displays the block table of SMX.

Without any arguments, a table with all created blocks will be shown. Specify a block magic number to display only one specific block.



“magic” is a unique ID, used by the OS Awareness to identify a specific block (address of the BCB).

The fields “address”, “pool” and “owner” are mouse sensitive, double clicking on them opens appropriate windows.

TASK.BUCKet

Display buckets

Format: **TASK.BUCKet** <bucket>

Displays the bucket table of SMX or detailed information about one specific bucket.

Without any arguments, a table with all created buckets will be shown.
Specify a bucket magic number to display detailed information on that bucket.

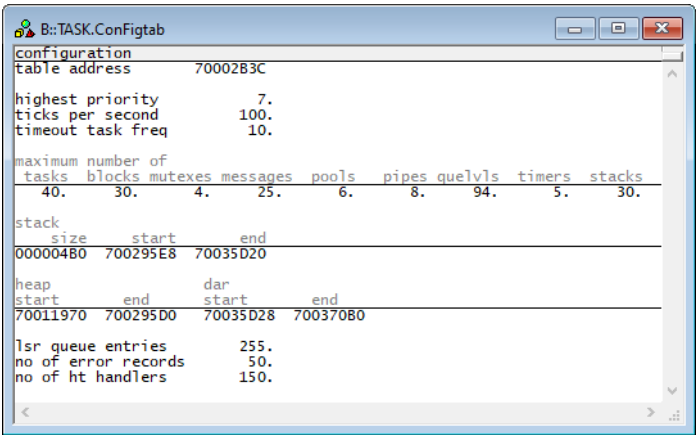
“magic” is a unique ID, used by the OS Awareness to identify a specific bucket (address of the BXCb).

The fields “magic”, “name”, “start”, “pointer” and “tasks waiting” are mouse sensitive, double clicking on them opens appropriate windows.

Format:

TASK.ConFigtab

Displays the configuration table of SMX.

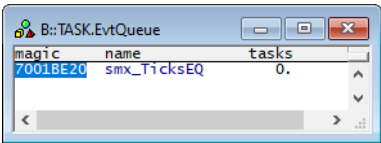


Format:

TASK.EvtQueue <eventqueue>

Displays the event queue table of SMX or detailed information about one specific event.

Without any arguments, a table with all created event queues will be shown. Specify an event magic number to display detailed information on that event.

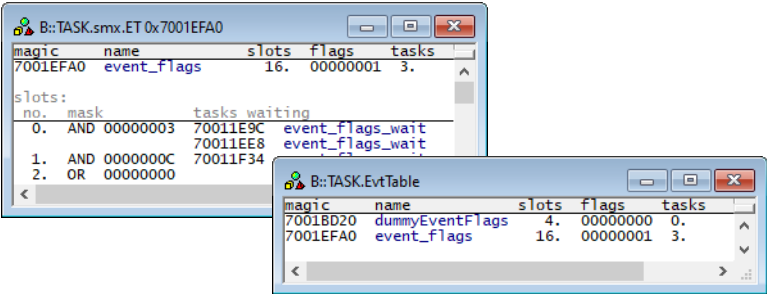


“magic” is a unique ID, used by the OS Awareness to identify a specific event (address of the ECB).
The fields “magic” and “name” are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.EvtTable** <eventtable>

Displays the event tables of SMX or detailed information about one specific event table.

Without any arguments, a table with all created event tables will be shown. Specify an event table magic number to display detailed information on that event table.



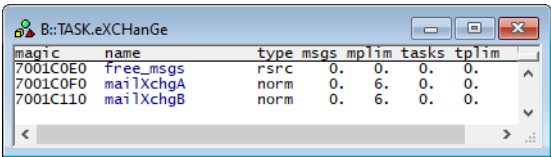
“magic” is a unique ID, used by the OS Awareness to identify a specific event table (address of the ETCB).

The fields “magic”, “name” and “tasks waiting” are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.eXCHanGe** <exchange>

Displays the exchange table of SMX or detailed information about one specific exchange.

Without any arguments, a table with all created exchanges will be shown. Specify an exchange magic number to display detailed information on that exchange.



The screenshot shows a window titled "B::TASK.eXCHanGe" containing a table with the following data:

magic	name	type	msgs	mplim	tasks	tplim
7001C0E0	free_msgs	rsrc	0.	0.	0.	0.
7001C0F0	mailXchgA	norm	0.	6.	0.	0.
7001C110	mailXchgB	norm	0.	6.	0.	0.

“magic” is a unique ID, used by the OS Awareness to identify a specific exchange (address of the XCB).

The fields “magic”, “name”, “address”, “resource” and “owner” are mouse sensitive, double clicking on them opens appropriate windows.

TASK.LSR

Display LSRs

Format: **TASK.LSR**

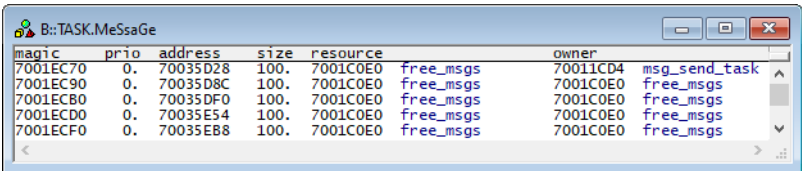
Displays the LSR table of SMX.

“magic” is a unique ID, used by the OS Awareness to identify a specific LSR (address of the LQ_CELL).

The field “entry” is mouse sensitive, double clicking on it opens the appropriate window.

Format: **TASK.MeSsaGe**

Displays the message table of SMX.



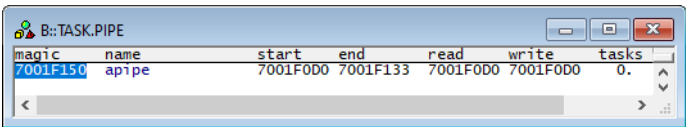
“magic” is a unique ID, used by the OS Awareness to identify a specific message (address of the MCB).

The fields “address”, “resource” and “owner” are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.PIPE** <pipe>

Displays the pipe table of SMX or detailed information about one specific pipe.

Without any arguments, a table with all created pipes will be shown. Specify a pipe magic number to display detailed information on that pipe.



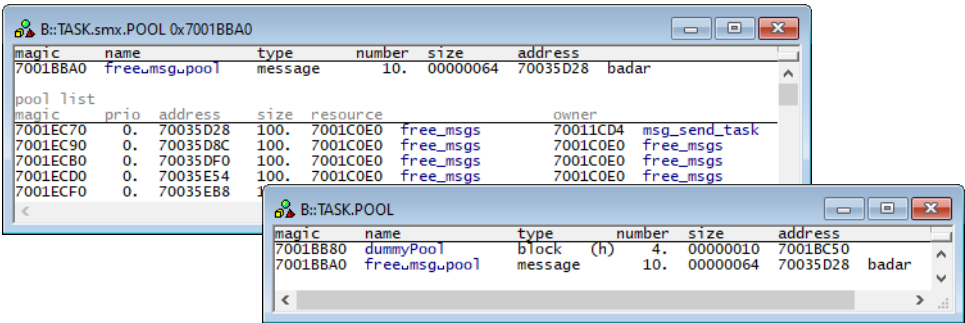
“magic” is a unique ID, used by the OS Awareness to identify a specific pipe (address of the PXCb).

The fields “magic”, “name”, “start”, “read” and “tasks waiting” are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.POOL** <pool>

Displays the pool table of SMX or detailed information about one specific pool.

Without any arguments, a table with all created pools will be shown. Specify a pool magic number to display detailed information on that pool.



“magic” is a unique ID, used by the OS Awareness to identify a specific pool (address of the PCB).

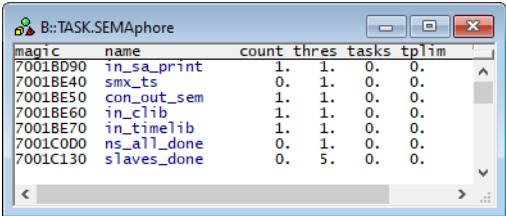
The fields “magic”, “name”, “address” and several fields in the pool list are mouse sensitive, double clicking on them opens appropriate windows.

Format:

TASK.SEMaphore <semaphore>

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

Without any arguments, a table with all created semaphores will be shown. Specify a semaphore magic number to display detailed information on that semaphore.



The screenshot shows a command window titled "B::TASK.SEMaphore". It displays a table with the following data:

magic	name	count	thres	tasks	tplim
7001BD90	in_sa_print	1.	1.	0.	0.
7001BE40	smx_ts	0.	1.	0.	0.
7001BE50	con_out_sem	1.	1.	0.	0.
7001BE60	in_clib	1.	1.	0.	0.
7001BE70	in_timelib	1.	1.	0.	0.
7001C0D0	ns_all_done	0.	1.	0.	0.
7001C130	slaves_done	0.	5.	0.	0.

“magic” is a unique ID, used by the OS Awareness to identify a specific semaphore (address of the SCB).

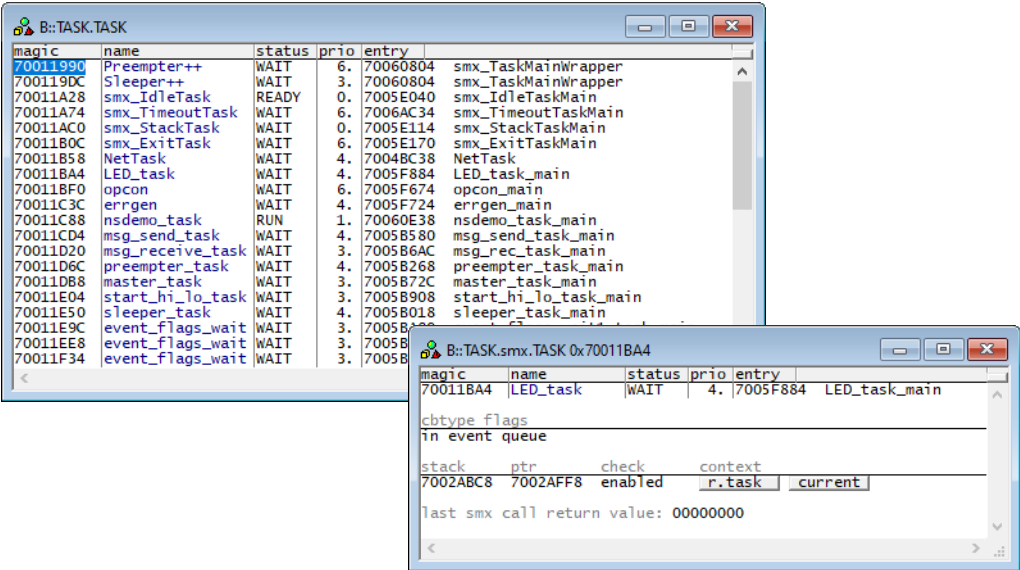
The fields “magic” and “name” are mouse sensitive, double clicking on them opens appropriate windows.

Format:

TASK.TASK <task>

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

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



- “magic” is a unique ID, used by the OS Awareness to identify a specific task (address of the TCB).
- “entry” shows either the task entry function, or the hook routine (if it is hooked).
- “stack” points to the block holding the stack; “ptr” is the stack pointer last saved by SMX.

The fields “magic”, “name”, “entry” and “stack” are mouse sensitive, double clicking on them opens appropriate windows.

Pressing the “r.task” button changes the register context to this task. “current” resets it to the current context. See “[Task Context Display](#)”.

TASK.TIMER

Display timers

Format:

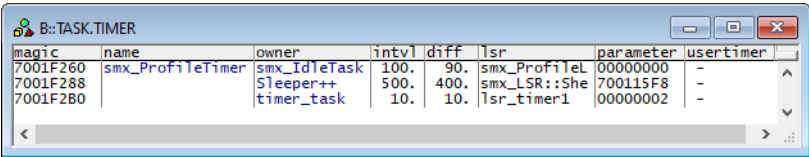
TASK.TIMER <timer>

Displays the timer table of SMX.

Without any arguments, a table with all created timers will be shown.
Specify a timer magic number to display only one specific timer.

“magic” is a unique ID, used by the OS Awareness to identify a specific timer (address of the TMCB).

The fields “magic”, “name”, “owner”, “lsr” and “usertimer” are mouse sensitive, double clicking on them opens appropriate windows.



magic	name	owner	intvl	diff	lsr	parameter	usertimer
7001F260	smx_ProfileTimer	smx_IdleTask	100.	90.	smx_ProfileL	00000000	-
7001F288		Sleeper++	500.	400.	smx_LSR::She	700115F8	-
7001F2B0		timer_task	10.	10.	lsr_timer1	00000002	-

TASK.TRACE

Display event buffer

Format:

TASK.TRACE

TASK.TRACE displays the kernel internal records of the event buffer feature.

SMX must be built with SMX_CFG_EVB. See SMX documentation more information on this SMX feature.

TASK.TRACEVM

Copy event buffer to LOGGER

Format:

TASK.TRACEVM

TASK.TRACEVM copies the entries of the kernel internal event buffer to a debugger-internal buffer in virtual memory (VM:), using the **LOGGER** structure layout and initializes the Logger. The **Logger.TimeStamp** is automatically set up by TASK.TRACEVM if possible, otherwise it must be set up explicitly.

SMX must be built with SMX_CFG_EVB. See SMX documentation more information on this SMX feature.

Activate the **LOGGER** and copy the buffers with:

```
Trace.METHOD Logger
Logger.RESet
TASK.TRACEVM
```

After this, you can use the Logger contents for [Task Runtime Statistics](#) and [Task State Analysis](#).

There are special definitions for SMX specific PRACTICE functions.

TASK.CONFIG()

OS Awareness configuration information

Syntax:

TASK.CONFIG(magic | magicsize)

Parameter and Description:

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

Return Value Type: Hex value.