LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual SMX

Release 09.2023

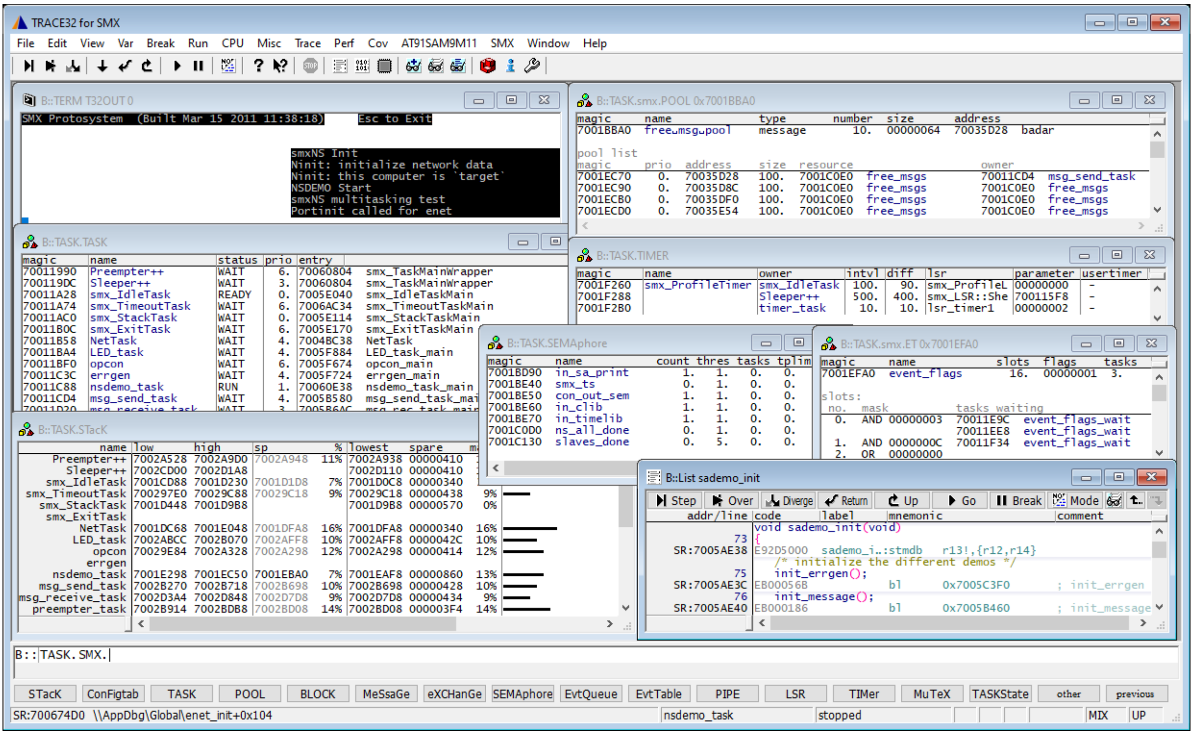
OS Awareness Manual SMX

Version 09-Oct-2023

History

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

Overview


TRACE32 for SMX
File Edit View Var Break Run CPU Misc Trace Perf Cov AT91SAM9M11 SMX Window Help
B::TERM T32OUT 0
B::TASK.smx.POOL 0x7001BBA0
B::TASK.TASK
B::TASK.TIMER
B::TASK.SEMAphore
B::TASK.smx.ET 0x7001EFA0
B::TASK.STacK
B::List sademo_init
Step Over Diverge Return Up Go Break Mode
STacK ConFigtab TASK POOL BLOCK MeSsaGe eXCHanGe SEMAphore EvtQueue EvtTable PIPE LSR TIMer MuTeX TASKState other previous

```
SMX Protosystem  (Built Mar 15 2011 11:38:18)      Esc to Exit

smxNS Init
Ninit: initialize network data
Ninit: this computer is `target`
NSDEMO Start
smxNS multitasking test
Portinit called for enet
```

```
magic     name              type        number  size      address
7001BBA0  free.msg.pool     message       10.   00000064  70035D28  badar

pool list
magic     prio  address   size  resource                    owner
7001EC70    0.  70035D28  100.  7001C0E0  free_msgs         70011CD4  msg_send_task
7001EC90    0.  70035D8C  100.  7001C0E0  free_msgs         7001C0E0  free_msgs
7001ECB0    0.  70035DF0  100.  7001C0E0  free_msgs         7001C0E0  free_msgs
7001ECD0    0.  70035E54  100.  7001C0E0  free_msgs         7001C0E0  free_msgs
```

```
magic     name              status prio entry
70011990  Preempter++       WAIT    6.  70060804  smx_TaskMainWrapper
700119DC  Sleeper++         WAIT    3.  70060804  smx_TaskMainWrapper
70011A28  smx_IdleTask      READY   0.  7005E040  smx_IdleTaskMain
70011A74  smx_TimeoutTask   WAIT    6.  7006AC34  smx_TimeoutTaskMain
70011AC0  smx_StackTask     WAIT    0.  7005E114  smx_StackTaskMain
70011B0C  smx_ExitTask      WAIT    6.  7005E170  smx_ExitTaskMain
70011B58  NetTask           WAIT    4.  7004BC38  NetTask
70011BA4  LED_task          WAIT    4.  7005F884  LED_task_main
70011BF0  opcon             WAIT    6.  7005F674  opcon_main
70011C3C  errgen            WAIT    4.  7005F724  errgen_main
70011C88  nsdemo_task       RUN     1.  70060E38  nsdemo_task_main
70011CD4  msg_send_task     WAIT    4.  7005B580  msg_send_task_mai
70011D20  msg_receive_task  WAIT    3.  7005B6AC  msg_rec_task_main
```

```
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   -
```

```
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     name            slots  flags     tasks
7001EFA0  event_flags       16.  00000001  3.

slots:
 no.  mask          tasks waiting
  0.  AND 00000003  70011E9C  event_flags_wait
                    70011EE8  event_flags_wait
  1.  AND 0000000C  70011F34  event_flags_wait
  2.  OR  00000000
```

```
            name  low      high     sp          %  lowest   spare     ma
     Preempter++  7002A528 7002A9D0 7002A948  11%  7002A938 00000410
       Sleeper++  7002CD00 7002D1A8                7002D110 00000410
    smx_IdleTask  7001CD88 7001D230 7001D1D8   7%  7001D0C8 00000340
 smx_TimeoutTask  700297E0 70029C88 70029C18   9%  70029C18 00000438   9%
   smx_StackTask  7001D448 7001D9B8                7001D9B8 00000570   0%
    smx_ExitTask
         NetTask  7001DC68 7001E048 7001DFA8  16%  7001DFA8 00000340  16%
        LED_task  7002ABCC 7002B070 7002AFF8  10%  7002AFF8 0000042C  10%
           opcon  70029E84 7002A328 7002A298  12%  7002A298 00000414  12%
          errgen
     nsdemo_task  7001E298 7001EC50 7001EBA0   7%  7001EAF8 00000860  13%
   msg_send_task  7002B270 7002B718 7002B698  10%  7002B698 00000428  10%
msg_receive_task  7002D3A4 7002D848 7002D7D8   9%  7002D7D8 00000434   9%
  preempter_task  7002B914 7002BDB8 7002BD08  14%  7002BD08 000003F4  14%
```

```
   addr/line code      label       mnemonic                  comment
               void sademo_init(void)
          73   {
 SR:7005AE38   E92D5000  sademo_i..:stmdb   r13!,{r12,r14}
                  /* initialize the different demos */
          75      init_errgen();
 SR:7005AE3C   EB00056B             bl        0x7005C3F0      ; init_errgen
          76      init_message();
 SR:7005AE40   EB000186             bl        0x7005B460      ; init_message
```

```
B::TASK.SMX.
```

```
SR:700674D0  \\AppDbg\Global\enet_init+0x104      nsdemo_task      stopped      MIX  UP
```

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 loose 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.

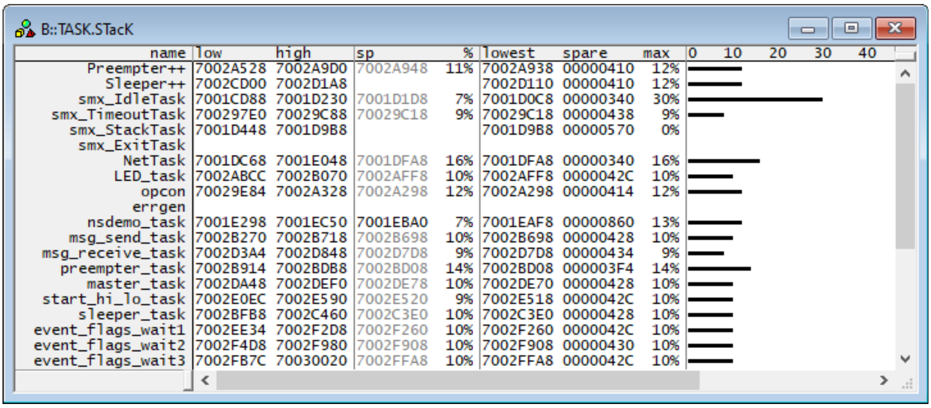
Task Stack Coverage

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.



B::TASK.STacK

name	low	high	sp	%	lowest	spare	max	0 10 20 30 40
Preempter++	7002A528	7002A9D0	7002A948	11%	7002A938	00000410	12%	
Sleeper++	7002CD00	7002D1A8			7002D110	00000410	12%	
smx_IdleTask	7001CD88	7001D230	7001D1D8	7%	7001D0C8	00000340	30%	
smx_TimeoutTask	700297E0	70029C88	70029C18	9%	70029C18	00000438	9%	
smx_StackTask	7001D448	7001D9B8			7001D9B8	00000570	0%	
smx_ExitTask								
NetTask	7001DC68	7001E048	7001DFA8	16%	7001DFA8	00000340	16%	
LED_task	7002ABCC	7002B070	7002AFF8	10%	7002AFF8	0000042C	10%	
opcon	70029E84	7002A328	7002A298	12%	7002A298	00000414	12%	
errgen								
nsdemo_task	7001E298	7001EC50	7001EBA0	7%	7001EAF8	00000860	13%	
msg_send_task	7002B270	7002B718	7002B698	10%	7002B698	00000428	10%	
msg_receive_task	7002D3A4	7002D848	7002D7D8	9%	7002D7D8	00000434	9%	
preempter_task	7002B914	7002BDB8	7002BD08	14%	7002BD08	000003F4	14%	
master_task	7002DA48	7002DEF0	7002DE78	10%	7002DE70	00000428	10%	
start_hi_lo_task	7002E0EC	7002E590	7002E520	9%	7002E518	0000042C	10%	
sleeper_task	7002BFB8	7002C460	7002C3E0	10%	7002C3E0	00000428	10%	
event_flags_wait1	7002EE34	7002F2D8	7002F260	10%	7002F260	0000042C	10%	
event_flags_wait2	7002F4D8	7002F980	7002F908	10%	7002F908	00000430	10%	
event_flags_wait3	7002FB7C	70030020	7002FFA8	10%	7002FFA8	0000042C	10%	

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.

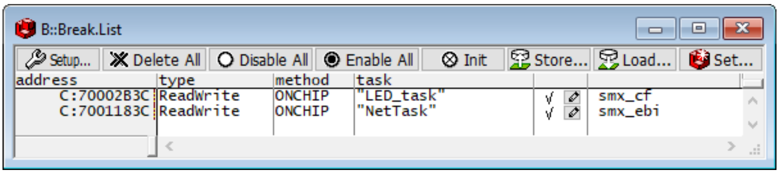
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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



```
address          type       method   task
    C:70002B3C  ReadWrite  ONCHIP   "LED_task"       √    smx_cf
    C:7001183C  ReadWrite  ONCHIP   "NetTask"        √    smx_ebi
```

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.

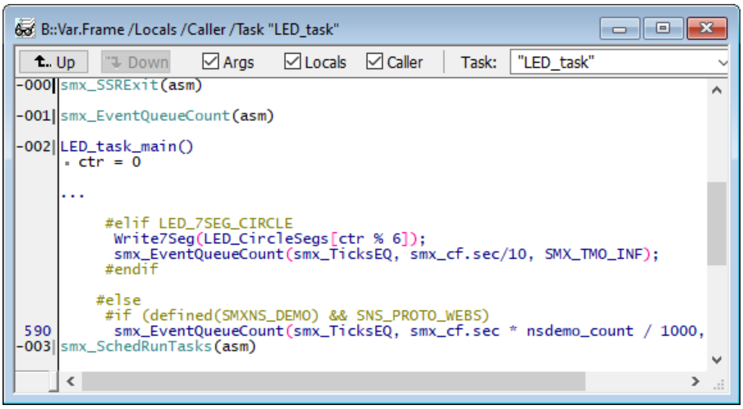
To display the call stack of a specific task, use the following command:

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.



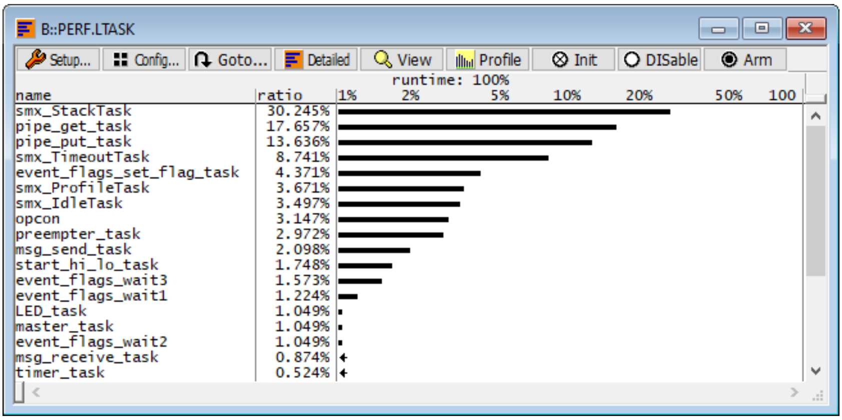
```
-000|smx_SSRExit(asm)

-001|smx_EventQueueCount(asm)

-002|LED_task_main()
    · ctr = 0

    ...

         #elif LED_7SEG_CIRCLE
          Write7Seg(LED_CircleSegs[ctr % 6]);
          smx_EventQueueCount(smx_TicksEQ, smx_cf.sec/10, SMX_TMO_INF);
         #endif

        #else
         #if (defined(SMXNS_DEMO) && SNS_PROTO_WEBS)
 590      smx_EventQueueCount(smx_TicksEQ, smx_cf.sec * nsdemo_count / 1000,
-003|smx_SchedRunTasks(asm)
```

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).



```
                                   runtime: 100%
name                     ratio   |1%     2%        5%      10%      20%        50%    100
smx_StackTask            30.245%
pipe_get_task            17.657%
pipe_put_task            13.636%
smx_TimeoutTask           8.741%
event_flags_set_flag_task 4.371%
smx_ProfileTask           3.671%
smx_IdleTask              3.497%
opcon                     3.147%
preempter_task            2.972%
msg_send_task             2.098%
start_hi_lo_task          1.748%
event_flags_wait3         1.573%
event_flags_wait1         1.224%
LED_task                  1.049%
master_task               1.049%
event_flags_wait2         1.049%
msg_receive_task          0.874%
timer_task                0.524%
```

Task Runtime Statistics

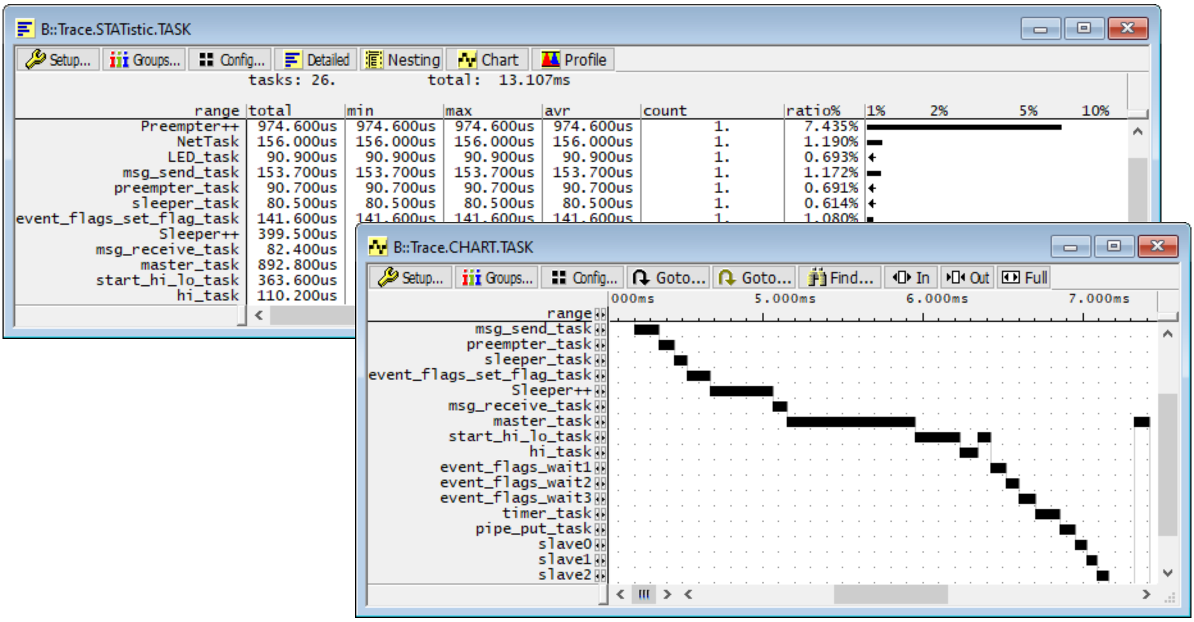
NOTE: This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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



```
                tasks: 26.          total:  13.107ms

                    range |total     |min       |max       |avr       |count              |ratio% |1%    2%       5%    10%
             Preempter++ | 974.600us| 974.600us| 974.600us| 974.600us|         1.         | 7.435%
                  NetTask| 156.000us| 156.000us| 156.000us| 156.000us|         1.         | 1.190%
                 LED_task|  90.900us|  90.900us|  90.900us|  90.900us|         1.         | 0.693%
            msg_send_task| 153.700us| 153.700us| 153.700us| 153.700us|         1.         | 1.172%
           preempter_task|  90.700us|  90.700us|  90.700us|  90.700us|         1.         | 0.691%
             sleeper_task|  80.500us|  80.500us|  80.500us|  80.500us|         1.         | 0.614%
event_flags_set_flag_task| 141.600us| 141.600us| 141.600us| 141.600us|         1.         | 1.080%
                Sleeper++| 399.500us
         msg_receive_task|  82.400us
              master_task| 892.800us
         start_hi_lo_task| 363.600us
                  hi_task| 110.200us
```

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
```

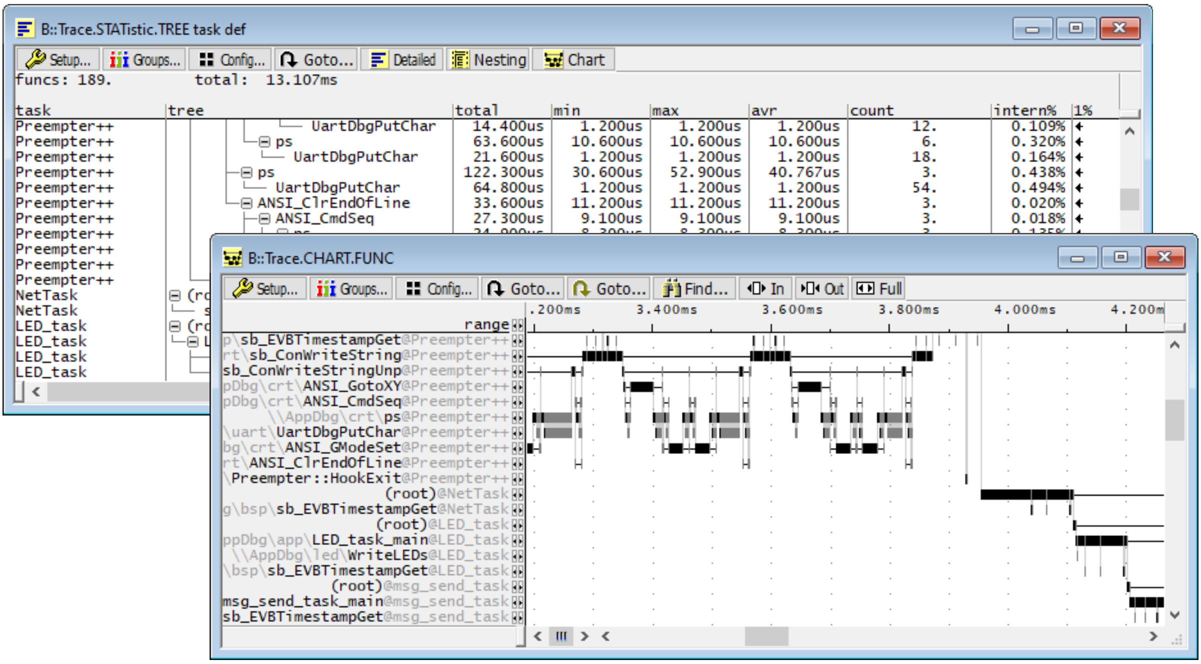
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)”.



```
funcs: 189.        total:  13.107ms

task         |tree                          |total     |min       |max       |avr       |count     |intern% |1%
Preempter++              └── UartDbgPutChar   14.400us   1.200us   1.200us   1.200us        12.   0.109%
Preempter++            ⊟ ps                   63.600us  10.600us  10.600us  10.600us         6.   0.320%
Preempter++              └── UartDbgPutChar   21.600us   1.200us   1.200us   1.200us        18.   0.164%
Preempter++          ⊟ ps                    122.300us  30.600us  52.900us  40.767us         3.   0.438%
Preempter++            └── UartDbgPutChar     64.800us   1.200us   1.200us   1.200us        54.   0.494%
Preempter++          ⊟ ANSI_ClrEndOfLine      33.600us  11.200us  11.200us  11.200us         3.   0.020%
Preempter++            ⊟ ANSI_CmdSeq          27.300us   9.100us   9.100us   9.100us         3.   0.018%
Preempter++
Preempter++
Preempter++
Preempter++
NetTask
NetTask
LED_task
LED_task
LED_task
LED_task
```

B::Trace.CHART.FUNC
Setup...
Groups...
Config...
Goto...
Goto...
Find...
In
Out
Full
.200ms
3.400ms
3.600ms
3.800ms
4.000ms
4.200m
range
p\sb_EVBTimestampGet@Preempter++
rt\sb_ConWriteString@Preempter++
sb_ConWriteStringUnp@Preempter++
pDbg\crt\ANSI_GotoXY@Preempter++
pDbg\crt\ANSI_CmdSeq@Preempter++
\\AppDbg\crt\ps@Preempter++
\uart\UartDbgPutChar@Preempter++
bg\crt\ANSI_GModeSet@Preempter++
rt\ANSI_ClrEndOfLine@Preempter++
\Preempter::HookExit@Preempter++
(root)@NetTask
g\bsp\sb_EVBTimestampGet@NetTask
(root)@LED_task
ppDbg\app\LED_task_main@LED_task
\\AppDbg\led\WriteLEDs@LED_task
\bsp\sb_EVBTimestampGet@LED_task
(root)@msg_send_task
msg_send_task_main@msg_send_task
sb_EVBTimestampGet@msg_send_task

SMX specific Menu

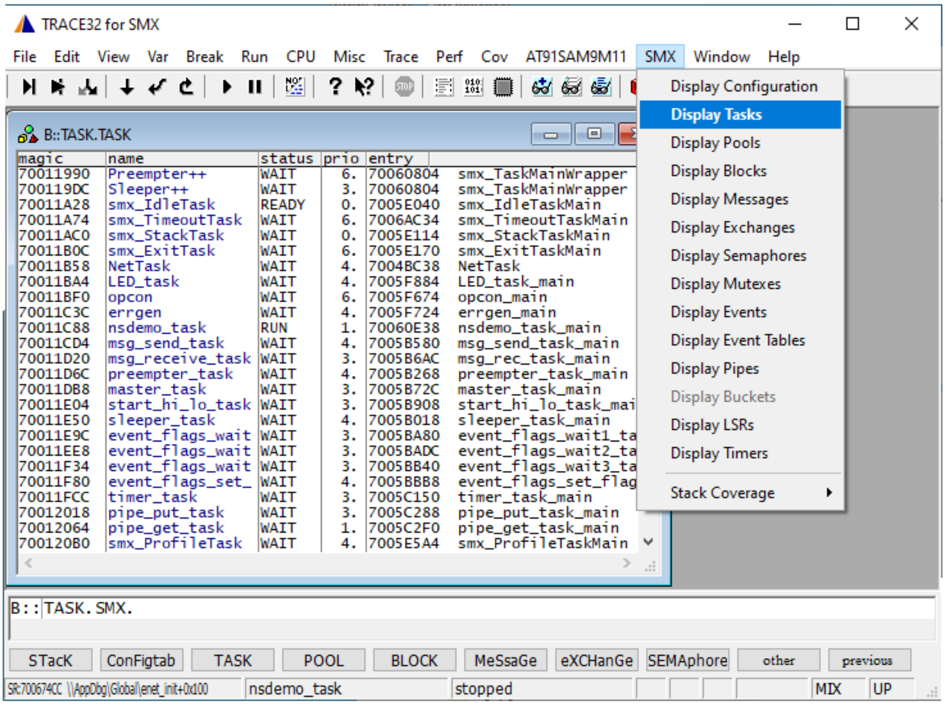
The menu file “smx.men” contains a menu with SMX specific menu items. Load this menu with the **MENU.ReProgram** command.

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.



```
magic    name              status prio entry
70011990 Preempter++       WAIT    6.  70060804  smx_TaskMainWrapper
700119DC Sleeper++         WAIT    3.  70060804  smx_TaskMainWrapper
70011A28 smx_IdleTask      READY   0.  7005E040  smx_IdleTaskMain
70011A74 smx_TimeoutTask   WAIT    6.  7006AC34  smx_TimeoutTaskMain
70011AC0 smx_StackTask     WAIT    0.  7005E114  smx_StackTaskMain
70011B0C smx_ExitTask      WAIT    6.  7005E170  smx_ExitTaskMain
70011B58 NetTask           WAIT    4.  7004BC38  NetTask
70011BA4 LED_task          WAIT    4.  7005F884  LED_task_main
70011BF0 opcon             WAIT    6.  7005F674  opcon_main
70011C3C errgen            WAIT    4.  7005F724  errgen_main
70011C88 nsdemo_task       RUN     1.  70060E38  nsdemo_task_main
70011CD4 msg_send_task     WAIT    4.  7005B580  msg_send_task_main
70011D20 msg_receive_task  WAIT    3.  7005B6AC  msg_rec_task_main
70011D6C preempter_task    WAIT    4.  7005B268  preempter_task_main
70011DB8 master_task       WAIT    3.  7005B72C  master_task_main
70011E04 start_hi_lo_task  WAIT    3.  7005B908  start_hi_lo_task_mai
70011E50 sleeper_task      WAIT    4.  7005B018  sleeper_task_main
70011E9C event_flags_wait  WAIT    3.  7005BA80  event_flags_wait1_ta
70011EE8 event_flags_wait  WAIT    3.  7005BADC  event_flags_wait2_ta
70011F34 event_flags_wait  WAIT    3.  7005BB40  event_flags_wait3_ta
70011F80 event_flags_set_  WAIT    4.  7005BBB8  event_flags_set_flag
70011FCC timer_task        WAIT    3.  7005C150  timer_task_main
70012018 pipe_put_task     WAIT    3.  7005C288  pipe_put_task_main
70012064 pipe_get_task     WAIT    1.  7005C2F0  pipe_get_task_main
700120B0 smx_ProfileTask   WAIT    4.  7005E5A4  smx_ProfileTaskMain
```

```
B::TASK.SMX.
```

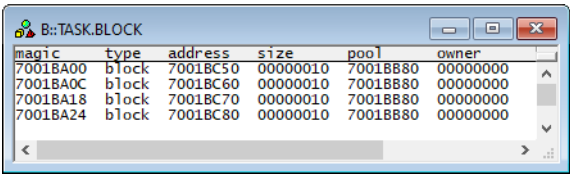
TASK.BLOCK

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     type   address   size      pool      owner
7001BA00  block  7001BC50  00000010  7001BB80  00000000
7001BA0C  block  7001BC60  00000010  7001BB80  00000000
7001BA18  block  7001BC70  00000010  7001BB80  00000000
7001BA24  block  7001BC80  00000010  7001BB80  00000000
```

“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.

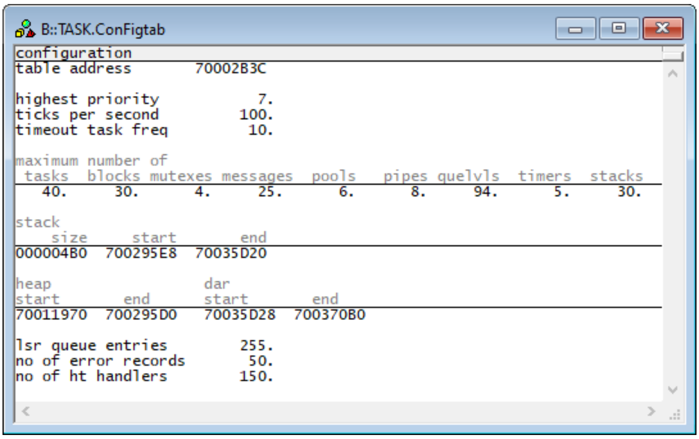
“magic” is a unique ID, used by the OS Awareness to identify a specific bucket (address of the BXCB).

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

TASK.ConFigtab Display configuration

Format: **TASK.ConFigtab**

Displays the configuration table of SMX.



B::TASK.ConFigtab

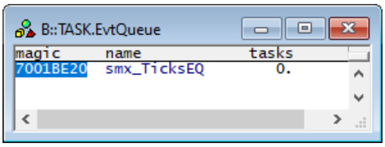
```
configuration
table address        70002B3C

highest priority           7.
ticks per second         100.
timeout task freq         10.

maximum number of
 tasks  blocks mutexes messages  pools   pipes quelvls  timers  stacks
   40.     30.      4.      25.     6.      8.     94.      5.     30.

stack
    size      start         end
000004B0  700295E8  70035D20

heap                dar
start        end    start        end
70011970  700295D0  70035D28  700370B0

lsr queue entries        255.
no of error records       50.
no of ht handlers        150.
```

TASK.EvtQueue Display event queues

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.



B::TASK.EvtQueue

```
magic     name            tasks
7001BE20  smx_TicksEQ        0.
```

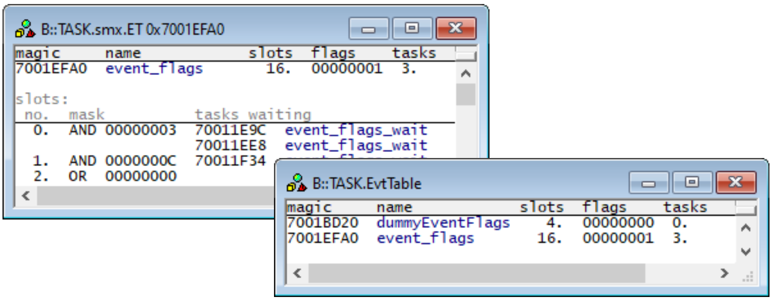
“magic” is a unique ID, used by the OS Awareness to identify a specific event (address of the ECB).

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.



B::TASK.smx.ET 0x7001EFA0

```
magic     name                slots  flags     tasks
7001EFA0  event_flags          16.   00000001  3.

slots:
 no.  mask                tasks waiting
  0.  AND  00000003   70011E9C  event_flags_wait
                      70011EE8  event_flags_wait
  1.  AND  0000000C   70011F34
  2.  OR   00000000
```

B::TASK.EvtTable

```
magic     name                slots  flags     tasks
7001BD20  dummyEventFlags      4.    00000000  0.
7001EFA0  event_flags         16.    00000001  3.
```

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

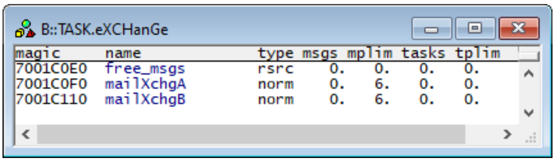
The fields “magic”, “name” and “tasks waiting” are mouse sensitive, double clicking on them opens appropriate windows.

TASK.eXCHanGe — Display exchanges

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.



```
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.

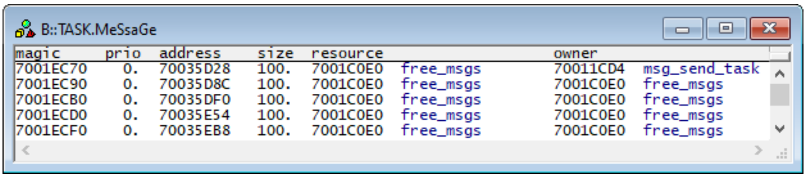
“magic” is a unique ID, used by the OS Awareness to identify a specific LSR (address of the LQ_CELL).

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

TASK.MeSsaGe Display messages

Format:	**TASK.MeSsaGe**

Displays the message table of SMX.



```
magic     prio  address   size  resource                     owner
7001EC70    0.  70035D28  100.  7001C0E0  free_msgs          70011CD4  msg_send_task
7001EC90    0.  70035D8C  100.  7001C0E0  free_msgs          7001C0E0  free_msgs
7001ECB0    0.  70035DF0  100.  7001C0E0  free_msgs          7001C0E0  free_msgs
7001ECD0    0.  70035E54  100.  7001C0E0  free_msgs          7001C0E0  free_msgs
7001ECF0    0.  70035EB8  100.  7001C0E0  free_msgs          7001C0E0  free_msgs
```

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

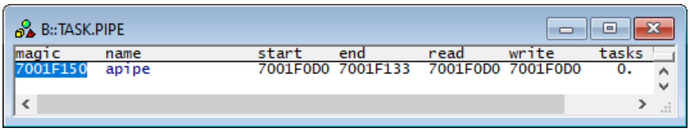
The fields “address”, “resource” and “owner” are mouse sensitive, double clicking on them opens appropriate windows.

TASK.PIPE Display pipes

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     name              start     end       read      write     tasks
7001F150  apipe             7001F0D0  7001F133  7001F0D0  7001F0D0     0.
```

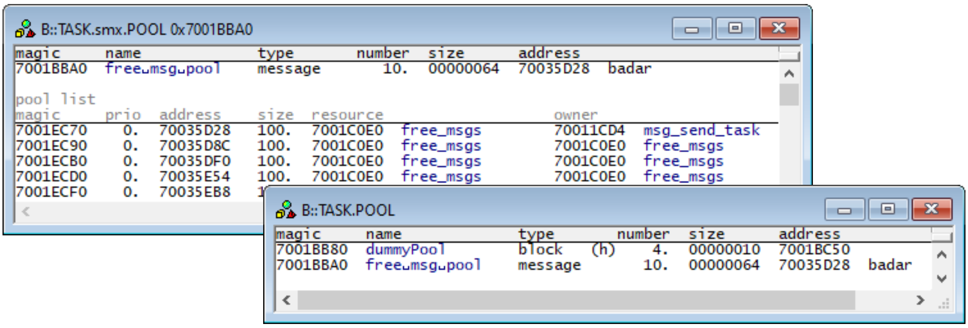
“magic” is a unique ID, used by the OS Awareness to identify a specific pipe (address of the PXCB).

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     name               type        number  size      address
7001BBA0  free␣msg␣pool      message        10.  00000064  70035D28  badar

pool list
magic     prio  address   size  resource                       owner
7001EC70    0.  70035D28  100.  7001C0E0  free_msgs            70011CD4  msg_send_task
7001EC90    0.  70035D8C  100.  7001C0E0  free_msgs            7001C0E0  free_msgs
7001ECB0    0.  70035DF0  100.  7001C0E0  free_msgs            7001C0E0  free_msgs
7001ECD0    0.  70035E54  100.  7001C0E0  free_msgs            7001C0E0  free_msgs
7001ECF0    0.  70035EB8  1
```

```
magic     name               type        number  size      address
7001BB80  dummyPool          block   (h)     4.  00000010  7001BC50
7001BBA0  free␣msg␣pool      message        10.  00000064  70035D28  badar
```

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

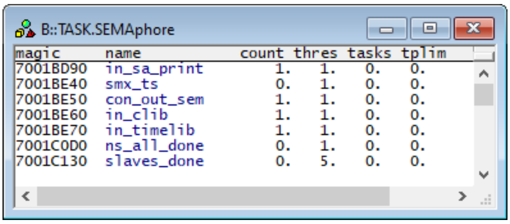
The fields “magic”, “name”, “address” and several fields in the pool list are mouse sensitive, double clicking on them opens appropriate windows.

TASK.SEMAphore

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.



```
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.

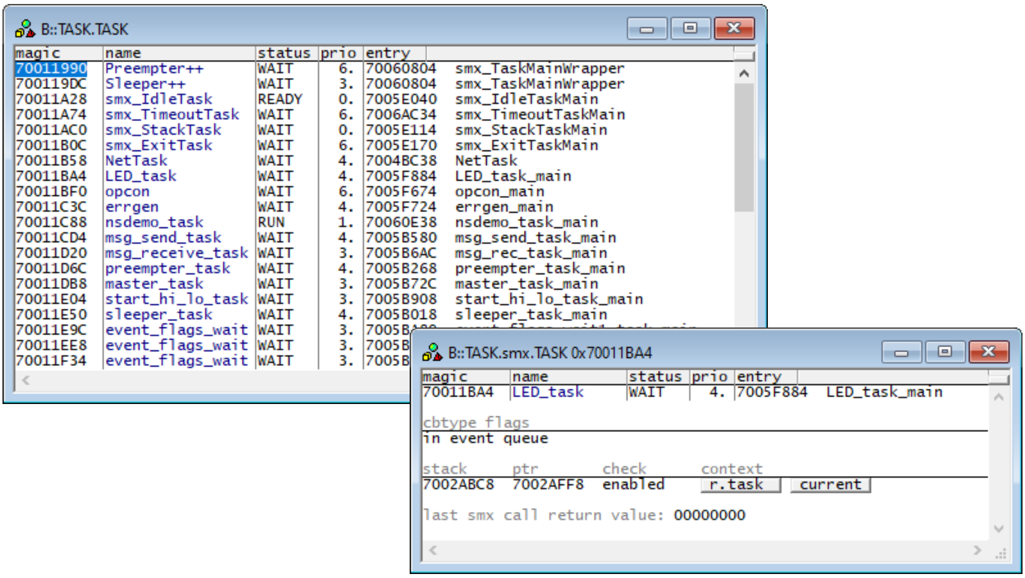
TASK.TASK

Display tasks

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.



```
B::TASK.TASK
magic     name              status prio entry
70011990  Preempter++       WAIT    6.  70060804  smx_TaskMainWrapper
700119DC  Sleeper++         WAIT    3.  70060804  smx_TaskMainWrapper
70011A28  smx_IdleTask      READY   0.  7005E040  smx_IdleTaskMain
70011A74  smx_TimeoutTask   WAIT    6.  7006AC34  smx_TimeoutTaskMain
70011AC0  smx_StackTask     WAIT    0.  7005E114  smx_StackTaskMain
70011B0C  smx_ExitTask      WAIT    6.  7005E170  smx_ExitTaskMain
70011B58  NetTask           WAIT    4.  7004BC38  NetTask
70011BA4  LED_task          WAIT    4.  7005F884  LED_task_main
70011BF0  opcon             WAIT    6.  7005F674  opcon_main
70011C3C  errgen            WAIT    4.  7005F724  errgen_main
70011C88  nsdemo_task       RUN     1.  70060E38  nsdemo_task_main
70011CD4  msg_send_task     WAIT    4.  7005B580  msg_send_task_main
70011D20  msg_receive_task  WAIT    3.  7005B6AC  msg_rec_task_main
70011D6C  preempter_task    WAIT    4.  7005B268  preempter_task_main
70011DB8  master_task       WAIT    3.  7005B72C  master_task_main
70011E04  start_hi_lo_task  WAIT    3.  7005B908  start_hi_lo_task_main
70011E50  sleeper_task      WAIT    4.  7005B018  sleeper_task_main
70011E9C  event_flags_wait  WAIT    3.  7005B
70011EE8  event_flags_wait  WAIT    3.  7005B
70011F34  event_flags_wait  WAIT    3.  7005B
```

```
B::TASK.smx.TASK 0x70011BA4
magic     name              status prio entry
70011BA4  LED_task          WAIT    4.  7005F884  LED_task_main

cbtype flags
in event queue

stack     ptr       check     context
7002ABC8  7002AFF8  enabled   r.task   current

last smx call return value: 00000000
```

"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

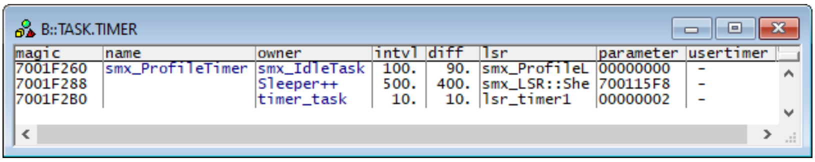
Format: **TASK.TIMer** <*timer*>

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**.

SMX PRACTICE Functions

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 (***without*** 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 (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.