# OS Awareness Manual QNX

# OS Awareness Manual QNX

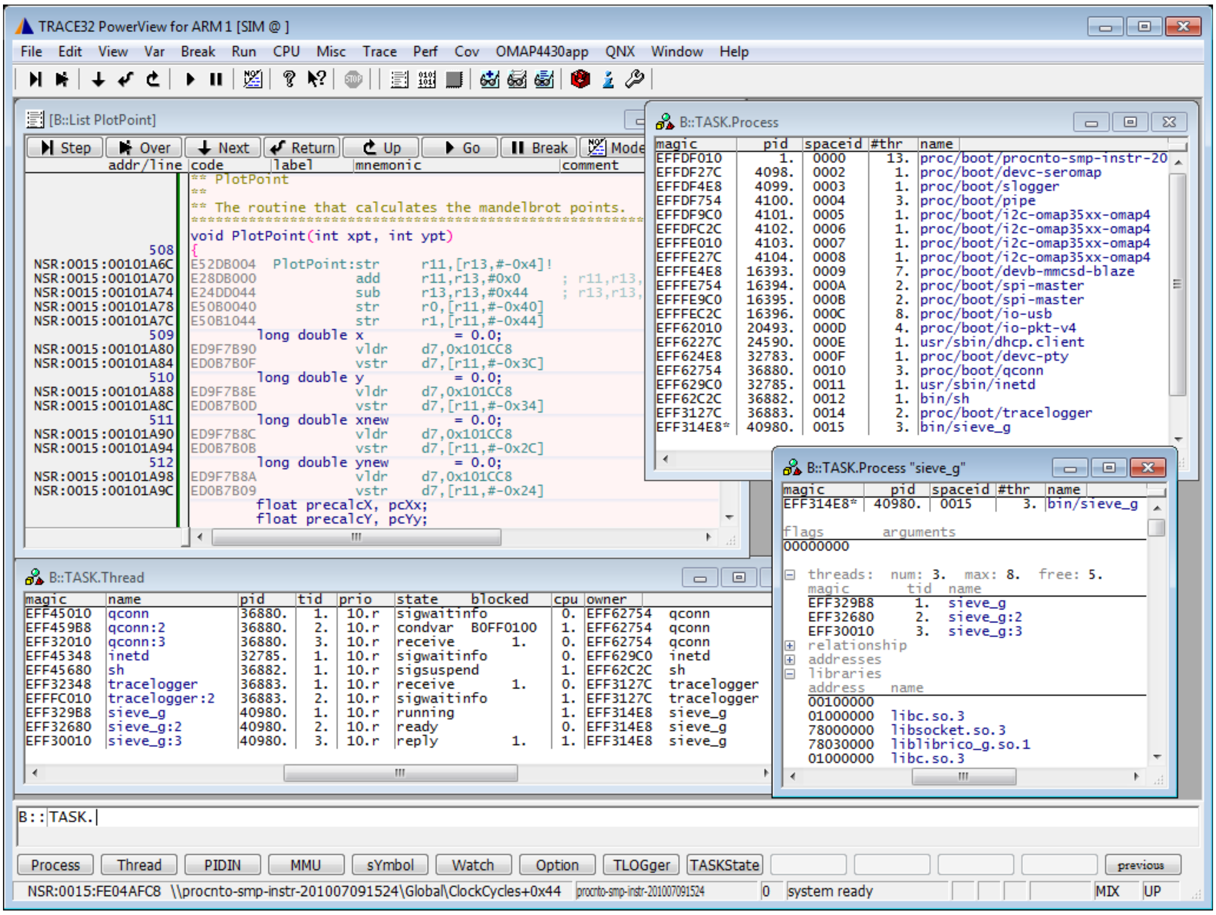
# OS Awareness Manual QNX

**Version 06-Jun-2024**

## History

| | |
|---|---|
| 07-Feb-2024 | New functions: TASK.QVM.NAME() and TASK.QVM.VMLIST(). |
| 04-Apr-2022 | Split the “Features” chapter into two chapters, “Debug Features” and “Trace Features”. |
| 04-Apr-2022 | Updated “Hooks & Internals in QNX” chapter. |
| 11-Mar-2022 | New functions: TASK.CORE.ASSIGN(), TASK.PROC.ID(), TASK.PROC.TTB(), TASK.ASINFO.START(), TASK.ASINFO.SIZE(). |
| 11-Mar-2022 | New functions: TASK.QVM.MAGIC(), TASK.QVM.MID(), and TASK.QVM.FORMAT(). |
| 10-Mar-2022 | New commands: TASK.IFS, TASK.SHMEM, TASK.SLOGGER2, and TASK.QVM. New option /**TTBHV** for the command TASK.Option. |

# Overview


TRACE32 PowerView for ARM 1 [SIM @ ]
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app QNX Window Help
[B::List PlotPoint]
Step Over Next Return Up Go Break Mode
B::TASK.Process
B::TASK.Process "sieve_g"
B::TASK.Thread
Process Thread PIDIN MMU sYmbol Watch Option TLOGger TASKState previous

```
           addr/line code      label           mnemonic                 comment
                     ** PlotPoint
                     **
                     ** The routine that calculates the mandelbrot points.
                     *******************************************************
                     void PlotPoint(int xpt, int ypt)
                 508 {
  NSR:0015:00101A6C E52DB004  PlotPoint:str    r11,[r13,#-0x4]!
  NSR:0015:00101A70 E28DB000            add    r11,r13,#0x0      ; r11,r13,
  NSR:0015:00101A74 E24DD044            sub    r13,r13,#0x44     ; r13,r13,
  NSR:0015:00101A78 E50B0040            str    r0,[r11,#-0x40]
  NSR:0015:00101A7C E50B1044            str    r1,[r11,#-0x44]
                 509          long double x          = 0.0;
  NSR:0015:00101A80 ED9F7B90            vldr   d7,0x101CC8
  NSR:0015:00101A84 ED0B7B0F            vstr   d7,[r11,#-0x3C]
                 510          long double y          = 0.0;
  NSR:0015:00101A88 ED9F7B8E            vldr   d7,0x101CC8
  NSR:0015:00101A8C ED0B7B0D            vstr   d7,[r11,#-0x34]
                 511          long double xnew       = 0.0;
  NSR:0015:00101A90 ED9F7B8C            vldr   d7,0x101CC8
  NSR:0015:00101A94 ED0B7B0B            vstr   d7,[r11,#-0x2C]
                 512          long double ynew       = 0.0;
  NSR:0015:00101A98 ED9F7B8A            vldr   d7,0x101CC8
  NSR:0015:00101A9C ED0B7B09            vstr   d7,[r11,#-0x24]
                              float precalcX, pcXx;
                              float precalcY, pcYy;
```

```
magic        pid  spaceid #thr name
EFFDF010      1.  0000    13. proc/boot/procnto-smp-instr-20
EFFDF27C   4098.  0002     1. proc/boot/devc-seromap
EFFDF4E8   4099.  0003     1. proc/boot/slogger
EFFDF754   4100.  0004     3. proc/boot/pipe
EFFDF9C0   4101.  0005     1. proc/boot/i2c-omap35xx-omap4
EFFDFC2C   4102.  0006     1. proc/boot/i2c-omap35xx-omap4
EFFFE010   4103.  0007     1. proc/boot/i2c-omap35xx-omap4
EFFFE27C   4104.  0008     1. proc/boot/i2c-omap35xx-omap4
EFFFE4E8  16393.  0009     7. proc/boot/devb-mmcsd-blaze
EFFFE754  16394.  000A     2. proc/boot/spi-master
EFFFE9C0  16395.  000B     2. proc/boot/spi-master
EFFFEC2C  16396.  000C     8. proc/boot/io-usb
EFF62010  20493.  000D     4. proc/boot/io-pkt-v4
EFF6227C  24590.  000E     1. usr/sbin/dhcp.client
EFF624E8  32783.  000F     1. proc/boot/devc-pty
EFF62754  36880.  0010     3. proc/boot/qconn
EFF629C0  32785.  0011     1. usr/sbin/inetd
EFF62C2C  36882.  0012     1. bin/sh
EFF3127C  36883.  0014     2. proc/boot/tracelogger
EFF314E8* 40980.  0015     3. bin/sieve_g
```

```
magic     name          pid     tid  prio  state     blocked  cpu owner
EFF45010  qconn         36880.  1.   10.r  sigwaitinfo        0.  EFF62754  qconn
EFF459B8  qconn:2       36880.  2.   10.r  condvar   B0FF0100 1.  EFF62754  qconn
EFF32010  qconn:3       36880.  3.   10.r  receive        1.  0.  EFF62754  qconn
EFF45348  inetd         32785.  1.   10.r  sigwaitinfo        0.  EFF629C0  inetd
EFF45680  sh            36882.  1.   10.r  sigsuspend         1.  EFF62C2C  sh
EFF32348  tracelogger   36883.  1.   10.r  receive        1.  0.  EFF3127C  tracelogger
EFFFC010  tracelogger:2 36883.  2.   10.r  sigwaitinfo        1.  EFF3127C  tracelogger
EFF329B8  sieve_g       40980.  1.   10.r  running            1.  EFF314E8  sieve_g
EFF32680  sieve_g:2     40980.  2.   10.r  ready              0.  EFF314E8  sieve_g
EFF30010  sieve_g:3     40980.  3.   10.r  reply          1.  1.  EFF314E8  sieve_g
```

```
magic        pid    spaceid #thr name
EFF314E8*  40980.   0015      3. bin/sieve_g

flags       arguments
00000000

⊟ threads:  num: 3.  max: 8.  free: 5.
  magic       tid  name
  EFF329B8     1.  sieve_g
  EFF32680     2.  sieve_g:2
  EFF30010     3.  sieve_g:3
⊞ relationship
⊞ addresses
⊟ libraries
  address   name
  00100000
  01000000  libc.so.3
  78000000  libsocket.so.3
  78030000  liblibrico_g.so.1
  01000000  libc.so.3
```

```
B::TASK.
```

```
NSR:0015:FE04AFC8  \\procnto-smp-instr-201007091524\Global\ClockCycles+0x44  procnto-smp-instr-201007091524  0  system ready  MIX  UP
```

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

## Terminology

QNX uses the terms “processes” and “threads”. If not otherwise specified, the TRACE32 term “task” corresponds to QNX threads.

## 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 QNX is supported for the following versions:

- QNX 6.1 to 6.6:
  - "armle" on ARM architectures,
  - "ppcbe" on PowerPC,
  - "shle" on SH4,
  - "x86" on Intel x86 architectures (32bit implementation)
- QNX 7.0, 7.1 and 8.0:
  - "armle-v7" on ARM32 architectures,
  - "aarch64le" on ARM64 architectures,
  - "x86" on Intel x86 architectures (32bit implementation)
  - "x86_64" on Intel x86 architectures (64bit implementation)

## Configuration

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

Automatic configuration tries to locate the QNX internals automatically. For this purpose all symbol tables 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).

For system resource display and trace 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 at any time, the OS Awareness is used. 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 ~~/demo/***<cpu>***/kernel/qnx/qnx.t32** |

(Note: “~~” refers to the TRACE32 installation directory)

Note that the kernel symbols from “procnto” must be loaded into the debugger. See **Hooks & Internals** for details on the used symbols.

See also the examples in the demo directories “~~/demo/*<cpu>*/kernel/qnx”.

## Quick Configuration Guide

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

1. Carefully read the demo start-up scripts (~~/demo/*<cpu>*/kernel/qnx).
2. Make a copy of the PRACTICE script “qnx.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 trace functions (if available).

Now you can access the QNX extensions through the menu.

In case of any problems, please carefully read the previous Configuration chapters.

## Hooks & Internals in QNX

No hooks are used in the kernel.

There are some requirements to do a successful debugging and tracing with QNX. In case of problems, please check carefully these items.

## Requirements for Debugging

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols of “procnto”. This means that every time, when features of the OS Awareness are used, the symbols of “procnto” must be available and accessible.

The system builder generates a linked symbol file called “procnto.sym” in the workspace’s “Images” directory, which needs to be loaded into the debugger.

**QNX 6.2:** To create the symbol file in your image directory, you need to add a line “`[+keeplinked]`” to your system build file.

**QNX 6.3/6.4:** To create the symbol file in your image directory, change in the System Builder Project (project.bld) the property “System -> Procnto/Startup Symbol Files” to “Yes”.

**QNX 6.5:** To create the symbol files in your image directory, open the System Builder Project (project.bld) and set the “System” properties “Create startup sym file?” and “Create proc sym file?” to “Yes”.

**QNX 6.6/7.0:** To create the symbol file in your image directory, you need to add a line “`[+keeplinked]`” to your system build file.

Please look at the demo startup script qnx.cmm, how to load the system symbols and the symbols of your application.

| | |
|---|---|
| **NOTE:** | In QNX version 6.5 and 6.6, the standard installation does not include debug information of the kernel, i.e. you will not be able to see the internal structures of a process or thread. The QNX awareness does not need this, so it’s sufficient to use the standard kernel. However, if you want access to these internal structures, you have to install and use the debug version - see **Appendix A**. |

## Requirements for Tracing

Tracing with QNX requires that the on-chip trace generation logic can generate process and/or thread information. For details refer to **“OS-aware Tracing”** (glossary.pdf).

On Arm architectures, QNX serves the ContextID register with the address space ID (ASID) of the process. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide trace information of threads.

To allow tracing of QNX threads, the context ID must contain the thread ID. See **Task Runtime Analysis** for an appropriate patch.

## Requirements for QNX Hypervisor

TRACE32 can be used to debug both, the QNX hypervisor host and any guest running as virtual machine (VM) within the hypervisor. In QNX, a VM is bound to a special QNX host process, called “qvm”. To be able to debug guests, the following requirements must be met:

- TRACE32 must be set up as a hypervisor debug environment.
- The QNX awareness for the QNX host must be set up completely.
- The symbols of the “qvm” process must be loaded. If there is more than one qvm process, it is enough to load the symbols of only one qvm.
- To be able to work with several qvm processes, the QNX host *must not* use ASLR, Ensure to start procnto with the switch “-m~r” to switch off address space layout randomization.

# Debug Features

The OS Awareness for QNX supports the following debug features.

## Display of Kernel Resources

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

| | |
|---|---|
| **TASK.Process** | Processes |
| **TASK.Thread** | Threads |
| **TASK.PIDIN** | pidin |
| **TASK.ASINFO** | address space information |
| **TASK.IFS** | IFS directory |
| **TASK.SHMEM** | shmem |
| **TASK.SLOGGER2** | slogger2 |
| **TASK.TLOGger** | tracelogger |
| **TASK.QVM** | VMs |

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

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

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

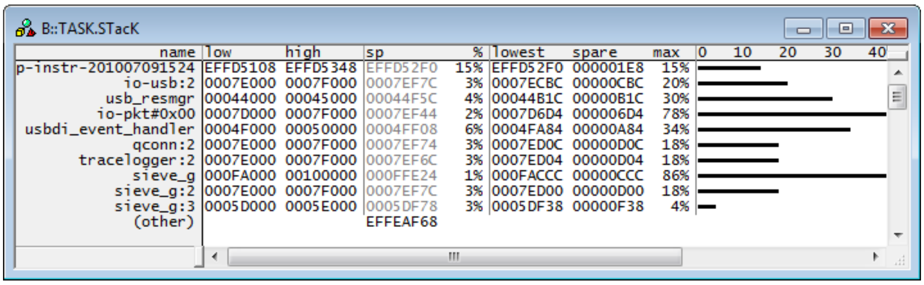
## Task Stack Coverage

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

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

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

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



```
                 name low      high     sp          % lowest   spare      max 0  10  20  30  40
p-instr-201007091524 EFFD5108 EFFD5348 EFFD52F0   15% EFFD52F0 000001E8   15%
            io-usb:2 0007E000 0007F000 0007EF7C    3% 0007ECBC 00000CBC   20%
          usb_resmgr 00044000 00045000 00044F5C    4% 00044B1C 00000B1C   30%
         io-pkt#0x00 0007D000 0007F000 0007EF44    2% 0007D6D4 000006D4   78%
  usbdi_event_handler 0004F000 00050000 0004FF08    6% 0004FA84 00000A84   34%
            qconn:2 0007E000 0007F000 0007EF74    3% 0007ED0C 00000D0C   18%
       tracelogger:2 0007E000 0007F000 0007EF6C    3% 0007ED04 00000D04   18%
             sieve_g 000FA000 00100000 000FFE24    1% 000FACCC 00000CCC   86%
           sieve_g:2 0007E000 0007F000 0007EF7C    3% 0007ED00 00000D00   18%
           sieve_g:3 0005D000 0005E000 0005DF78    3% 0005DF38 00000F38    4%
             (other)                   EFFEAF68
```

| **NOTE:** | The stack coverage **only** evaluates the stack area that is currently mapped into the MMU of the process. While running, QNX may map additional pages to the stack. QNX usually does not initialize the stack before use. Thus the maximum stack usage may show wrong results. |
|---|---|

## 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.

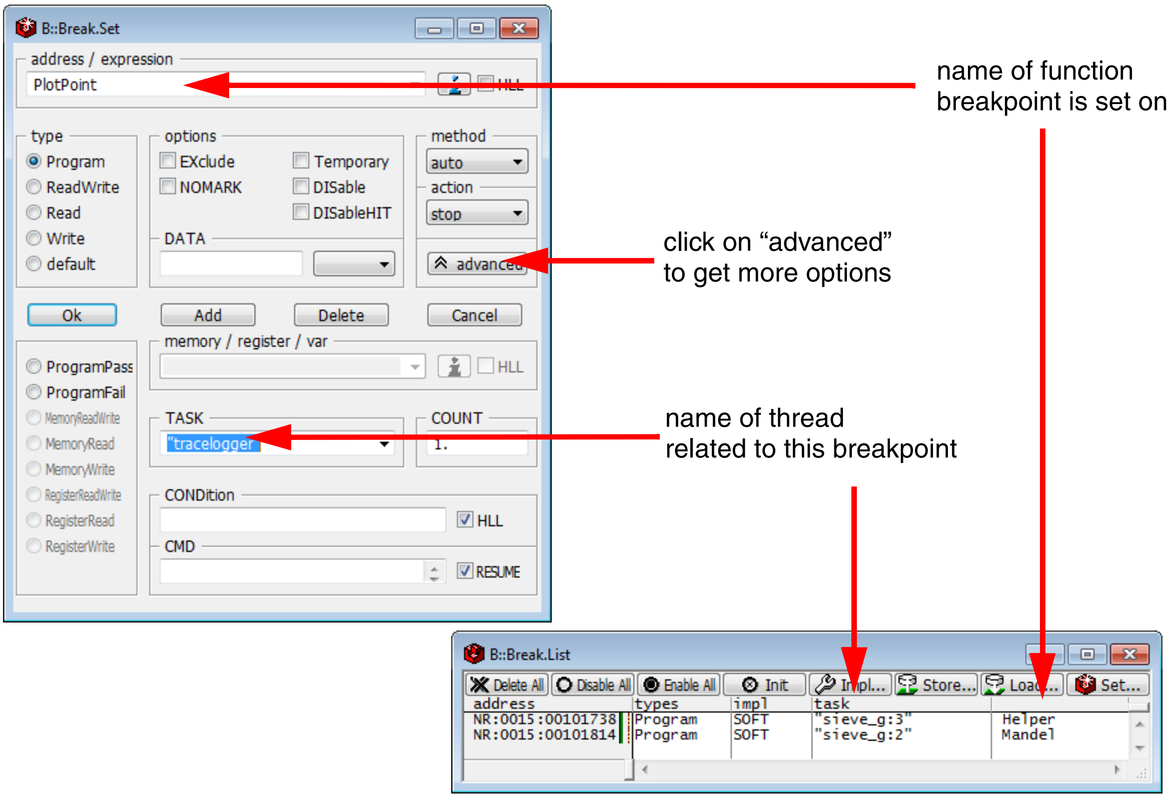
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

| | |
|---|---|
| **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.

Example for a task-related breakpoint, equivalent to the **Break.Set <function> /TASK <task>** command:



```
address / expression: PlotPoint
type: Program
options: EXclude [ ]  Temporary [ ]  NOMARK [ ]  DISable [ ]  DISableHIT [ ]
method: auto
action: stop
TASK: "tracelogger"
COUNT: 1.
CONDition: HLL [✓]
CMD: RESUME [✓]
```

```
address              types    impl  task           
NR:0015:00101738     Program  SOFT  "sieve_g:3"    Helper
NR:0015:00101814     Program  SOFT  "sieve_g:2"    Mandel
```

## 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.

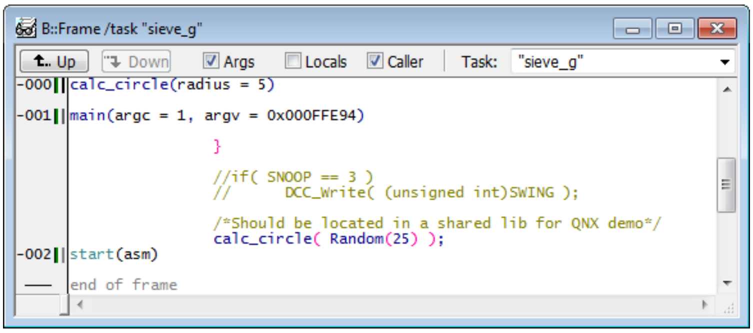
To display the call stack of a specific task, use the following command:

| | |
|---|---|
| **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 calc_circle(radius = 5)

-001 main(argc = 1, argv = 0x000FFE94)
                    }

                    //if( SNOOP == 3 )
                    //      DCC_Write( (unsigned int)SWING );

                    /*Should be located in a shared lib for QNX demo*/
                    calc_circle( Random(25) );
-002 start(asm)

 ——  end of frame
```

# MMU Support

To provide full debugging possibilities, the Debugger has to know, how virtual addresses are translated to physical addresses and vice versa. All **MMU** and **TRANSlation** commands refer to this necessity.

## Space IDs

Different processes of QNX may use identical virtual addresses. To distinguish those addresses, the debugger uses an additional identifier, the so-called space ID (memory space ID) that specifies, to which virtual memory space the address refers to. The command **SYStem.Option.MMUSPACES ON** enables the use of the space ID. For all processes using the kernel address space, the space ID is zero. For processes using their own address space, the space ID corresponds to the process ID (but is not equal to). Threads of a particular process use the memory space of the invoking parent process. Consequently threads have the same space ID as the parent process.

You may scan the whole system for space IDs using the command **TRANSlation.ScanID**. Use **TRANSlation.ListID** to get a list of all recognized space IDs.

The function **task.proc.space("<process>")** returns the space ID for a given process.

## MMU Declaration

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

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

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

**<format> Options for ARM:**

| <format> | Description |
|---|---|
| **QNX.PLAIN** | QNX format using the ARM FCSE translation. Use this format only if the kernel address range starts at a lower addresses than 0xFC000000. Other than format QNX.fcse, page table entries in the range 0x02000000 <= VA < 0xFC000000 are not hidden, but MMU.List.PageTable shows valid translations between 0x02000000 and the begin of the kernel address space which are actually not used by the OS. */ |
| **QNX.fcse** | Standard QNX format using the ARM FCSE translation, assuming a kernel address range of 0xFC000000--0xFFFFFFFF. Page table entries for 0x02000000 <= VA < 0xFC000000 are hidden because these are neither process nor kernel specific addresses. */ |
| **STD** | Standard format defined by the CPU |
| **TINY** | MMU format using a tiny page size of only 1024 bytes |

**<format> Options for PowerPC:**

| <format> | Description |
|---|---|
| **QNX** | QNX standard format |
| **QNXBIG** | QNX format with 64-bit table entries<br>(QNX 6.4/6.5 at booke and 900 cores). Covers 32-bit virtual address range. |
| **STD** | Standard format defined by the CPU |

## <format> Options for RISC-V:

| <format> | Description |
|---|---|
| **STD** | Automatic detection of the page table format from the SATP register. |
| **SV32** | 32-bit page table format (for SV32 targets only) |
| **SV32X4** | Stage 2 (G-stage) 32-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV39** | 39-bit page table format (for SV64 targets only) |
| **SV39X4** | Stage 2 (G-stage) 39-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV48** | 48-bit page table format (for SV64 targets only) |
| **SV48X4** | Stage 2 (G-stage) 48-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV57** | 57-bit page table format (for SV64 targets only) |
| **SV57X4** | Stage 2 (G-stage) 57-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |

## <format> Options for SH4:

| <format> | Description |
|---|---|
| **QNX** | QNX standard format |

## <format> Options for x86:

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

### <base_address>

*<base_address>* is currently unused. Specify zero.

### <logical_kernel_address_range>

*<logical_kernel_address_range>* specifies the virtual to physical address translation of the kernel address range. Typically the kernel has a 1:1 translation.

### <physical_kernel_address>

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

When declaring the MMU layout, you should also create the kernel translation manually with **TRANSlation.Create**.

The kernel code, which resides in the kernel space, can be accessed by any process, regardless of the current space ID. Use the command **TRANSlation.COMMON** to define the complete address range that is addressed by the kernel as commonly used area.

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

Setting up the MMU declaration is highly architecture and system dependent, please see the example scripts in the ~~/demo directory.

## Scanning System and Processes

To access the different process spaces correctly, the debugger needs to know the address translation of every virtual address it uses. You can either scan the MMU tables and place a copy of them into the debugger's address translation table, or you can use a table walk, where the debugger walks through the MMU tables each time it accesses a virtual address.

The command **MMU.SCAN** *only* scans the contents of the current processor MMU settings. Use the command **MMU.SCAN ALL** to go through all space IDs and scan their MMU settings. Note that on some systems, this may take a long time. In this case you may scan single processes (see below).

The MMU of the SH4 has an address translation that cannot be scanned fully automatically. However, the current used memory areas can be scanned with **MMU.SCAN UTLB** and **MMU.SCAN ITLB**.

To scan the address translation of a specific process, use the command **MMU.SCAN TaskPageTable** *<process>*. This command scans the space ID of the specified process. To scan the kernel space, use:

```
MMU.SCAN TaskPageTable "procnto"
```

**TRANSlation.List** shows the address translation table for all scanned space IDs.

If you set **TRANSlation.TableWalk ON**, the debugger tries first to look up the address translation in it's own table (**TRANSlation.List**). If this fails, it walks through the target MMU tables to find the translation for a specific address. This feature eliminates the need of scanning the MMU each time it changed, but walking through the tables for each address may result in a very slow reading of the target. The address translations found with the table walk are only temporarily valid (i.e. not stored in **TRANSlation.List**), and are invalidated at each **Go** or **Step**.

See also chapter “**Debugging QNX Kernel and User Processes**”.

## Symbol Autoloader

The OS Awareness for QNX contains a “Symbol Autoloader”, which automatically loads symbol files corresponding to executed processes or libraries. The autoloader maintains a list of address ranges, corresponding to QNX components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader (e.g. via **Data.List**), the debugger invokes the command necessary to load the corresponding symbols to the appropriate addresses (including relocation). This is usually done via a PRACTICE script.

In order to load symbol files, the debugger needs to be aware of the currently loaded components. This information is available in the kernel data structures and can be interpreted by the debugger. The command **sYmbol.AutoLOAD.CHECK** defines, *when* these kernel data structures are read by the debugger (only on demand or after each program execution).

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

The loaded components can change over time, when processes are started and stopped and libraries are loaded or unloaded. The command **sYmbol.AutoLOAD.CHECK** configures the strategy, when to “check” the kernel data structures for changes in order to keep the debugger’s information regarding the components up-to-date.

*Without parameters*, the **sYmbol.AutoLOAD.CHECK** command *immediately* updates the component information by reading the kernel data structures. This information includes the component name, the load address and the space ID and is used to fill the autoloader list (shown via **sYmbol.AutoLOAD.List**).

With **sYmbol.AutoLOAD.CHECK ON**, the debugger *automatically* reads the component information *each time the target stops executing* (even after assembly steps), having to assume that the component information might have changed. This significantly slows down the debugger which is inconvenient and often superfluous, e.g. when stepping through code that does not load or unload components.

With the parameter **ONGO**, the debugger checks for changed component info like with `ON`, but *not when performing single steps*.

With **sYmbol.AutoLOAD.CHECK OFF**, no automatic read is performed. In this case, the update has to be triggered manually when considered necessary by the user.

**NOTE:** The autoloader covers only components that are already started. Components that are not in the current process or library table are not covered.

The command **TASK.sYmbol.Option AutoLoad** configures which types of components the autoloader shall consider:

- Processes,
- All libraries, or
- Libraries of the current process.

It is recommended to restrict the components to the minimal set of interest (rather than all components), because it makes the autoloader checks much faster. By default, only processes are checked by the autoloader.

When configuring the OS Awareness for QNX, set up the symbol autoloader with the following command:

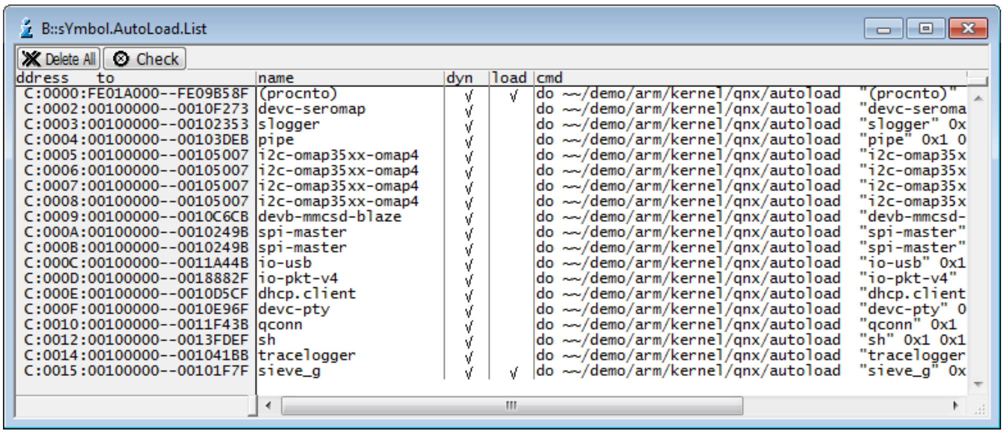
| | |
|---|---|
| Format: | **sYmbol.AutoLOAD.CHECKQNX** "*<action>*" |
| *<action>*: | action to take for symbol load, e.g.: `"do autoload"` |

The command **sYmbol.AutoLOAD.CHECKQNX** is used to define which action is to be taken, for loading the symbols corresponding to a specific address. The action defined is invoked with specific parameters (see below). With QnX, the pre-defined action is to call the script ~~/demo/*<cpu>*/kernel/qnx/autoload.cmm.

Note: the action parameter needs to be written with quotation marks (for the parser it is a string).

Note that *defining* this action, does not cause its execution. The action is executed on demand, i.e. when the address is actually accessed by the debugger e.g. in the **Data.List** or **Trace.List** window. In this case the autoloader executes the *<action>* appending parameters indicating the name of the component, its type (process, library), the load address and space ID.

For checking the currently active components use the command **sYmbol.AutoLOAD.List**. Together with the component name, it shows details like the load address, the space ID, and the command that will be executed to load the corresponding object files with symbol information. Only components shown in this list are handled by the autoloader.



```
ddress    to                  name                 dyn  load  cmd
 C:0000:FE01A000--FE09B58F   (procnto)             √     √    do ~~/demo/arm/kernel/qnx/autoload  "(procnto)"
 C:0002:00100000--0010F273   devc-seromap          √          do ~~/demo/arm/kernel/qnx/autoload  "devc-seroma
 C:0003:00100000--00102353   slogger               √          do ~~/demo/arm/kernel/qnx/autoload  "slogger" 0x
 C:0004:00100000--00103DEB   pipe                  √          do ~~/demo/arm/kernel/qnx/autoload  "pipe" 0x1 0
 C:0005:00100000--00105007   i2c-omap35xx-omap4    √          do ~~/demo/arm/kernel/qnx/autoload  "i2c-omap35x
 C:0006:00100000--00105007   i2c-omap35xx-omap4    √          do ~~/demo/arm/kernel/qnx/autoload  "i2c-omap35x
 C:0007:00100000--00105007   i2c-omap35xx-omap4    √          do ~~/demo/arm/kernel/qnx/autoload  "i2c-omap35x
 C:0008:00100000--00105007   i2c-omap35xx-omap4    √          do ~~/demo/arm/kernel/qnx/autoload  "i2c-omap35x
 C:0009:00100000--0010C6CB   devb-mmcsd-blaze      √          do ~~/demo/arm/kernel/qnx/autoload  "devb-mmcsd-
 C:000A:00100000--0010249B   spi-master            √          do ~~/demo/arm/kernel/qnx/autoload  "spi-master"
 C:000B:00100000--0010249B   spi-master            √          do ~~/demo/arm/kernel/qnx/autoload  "spi-master"
 C:000C:00100000--0011A44B   io-usb                √          do ~~/demo/arm/kernel/qnx/autoload  "io-usb" 0x1
 C:000D:00100000--0018882F   io-pkt-v4             √          do ~~/demo/arm/kernel/qnx/autoload  "io-pkt-v4"
 C:000E:00100000--0010D5CF   dhcp.client           √          do ~~/demo/arm/kernel/qnx/autoload  "dhcp.client
 C:000F:00100000--0010E96F   devc-pty              √          do ~~/demo/arm/kernel/qnx/autoload  "devc-pty" 0
 C:0010:00100000--0011F43B   qconn                 √          do ~~/demo/arm/kernel/qnx/autoload  "qconn" 0x1
 C:0012:00100000--0013FDEF   sh                    √          do ~~/demo/arm/kernel/qnx/autoload  "sh" 0x1 0x1
 C:0014:00100000--001041BB   tracelogger           √          do ~~/demo/arm/kernel/qnx/autoload  "tracelogger
 C:0015:00100000--00101F7F   sieve_g               √     √    do ~~/demo/arm/kernel/qnx/autoload  "sieve_g" 0x
```

| | |
|---|---|
| **NOTE:** | The GNU compiler generates different code if an application is built with debug info (option “-g”), even if the optimization level is the same. Ensure that you *always* use the debug version on *both* sides, the target where you start the application, and the debugger where you load the symbol file. |

## SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

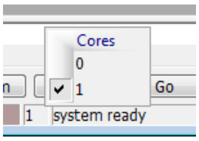
An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application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



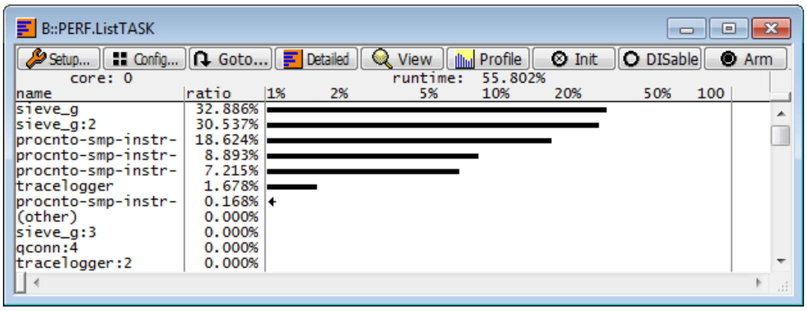
In SMP systems, the **TASK.Thread** command contains a “cpu” column that shows at which core the task is running, or was running the last time.

## 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).

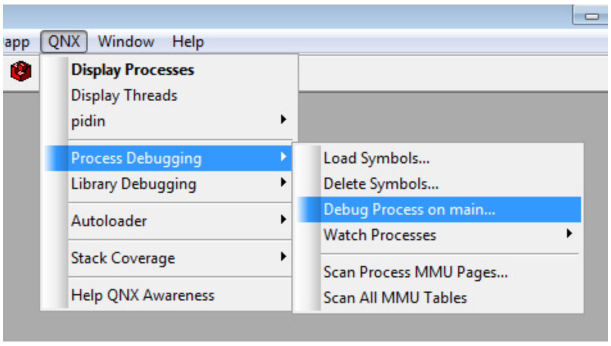


```
     core: 0                          runtime:   55.802%
name                ratio    1%     2%       5%     10%     20%      50%    100
sieve_g             32.886%
sieve_g:2           30.537%
procnto-smp-instr-  18.624%
procnto-smp-instr-   8.893%
procnto-smp-instr-   7.215%
tracelogger          1.678%
procnto-smp-instr-   0.168%
(other)              0.000%
sieve_g:3            0.000%
qconn:4              0.000%
tracelogger:2        0.000%
```

## QNX specific Menu

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

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



- The **Display** menu items launch the kernel resource display windows. See chapter “**Display of Kernel Resources**”.
- **Process Debugging** refers to actions related to process based debugging.
  See also chapter “**Debugging the Process**”.
  - Use **Load Symbols** and **Delete Symbols** to load rsp. delete the symbols of a specific process. See also **TASK.sYmbol**.
  - **Debug Process on main** allows you to start debugging a process on it’s main() function. Select this prior to starting the process. Specify the name of the process you want to debug. Then start the process in your terminal. The debugger will load the symbols and halt at main(). See also the demo script “app_debug.cmm”.
  - **Watch Processes** opens a process watch window or adds or removes processes from the process watch window. Specify a process name. See **TASK.Watch** for details.
  - **Scan Process MMU Pages** scans the MMU pages of the specified process.
  - **Scan All MMU Tables** performs a scan over all target side kernel and process MMU pages. See also chapter “**Scanning System and Processes**”.
- **“Library Debugging”** refers to actions related to library based debugging.
  See also chapter “**Debugging into Shared Libraries**”.
  - Use **Load Symbols** and **Delete Symbols** to load rsp. delete the symbols of a specific library. Please specify the library name and the process name that uses this library. You may select a symbol file on the host with the **Browse** button. See also **TASK.sYmbol**.
  - **Scan Process MMU Pages** scans the MMU pages of the specified process. Specify the name of the process that uses the library you want to debug.
    **Scan All MMU Tables** performs a scan over all target side kernel and process MMU pages. See also chapter “**Scanning System and Processes**”.
- Use the **Autoloader** submenu to configure the symbol autoloader.
  See also chapter “**Symbol Autoloader**”.
  - **List Components** opens a **sYmbol.AutoLOAD.List** window showing all components currently active in the autoloader.

  - **Check Now!** performs a **sYmbol.AutoLOAD.CHECK** and reloads the autoloader list.
  - **Set Loader Script** allows you to specify the script that is called when a symbol file load is required. You may also set the automatic autoloader check.
  - Use **Set Components Checked** to specify, which QNX components should be managed by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.

- The **Stack Coverage** submenu starts and resets the QNX specific stack coverage and provides an easy way to add or remove threads from the stack coverage window.

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

- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only thread switches (if any) or thread switches together with the default display.

The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime statistics or statistics on task states, if a trace is available. See also chapter “**Task Runtime Statistics**”.

# Trace Features

The OS Awareness for QNX 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)".

All kernel activities up to the thread switch are added to the calling thread.
If a process or thread terminates *before* the trace is evaluated, the debugger cannot assign a correct name to it. Instead the debugger will show a hex value for this process/thread.

On ARM architectures, QNX serves the ContextID register with the address space ID (ASID) of the process. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide performance information of threads.

To allow a detailed performance analysis on QNX threads, the context ID must contain the thread ID. Set the lower 8 bit of the context ID register with the process' ASID, and set the upper 24 bit with the lower 24bit of the address of the thread entry, i.e. "`(thread << 8) | ASID`".

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

**TASK.Option THRCTX ON**

To implement the above context ID setting, either patch the kernel or implement a "kerop_microaccount_hook". Ask Lauterbach for support if you need assistance.

## 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)”.

If a process or thread terminates *before* the trace is evaluated, the debugger cannot assign a correct name to it. Instead the debugger will show a hex value for this process/thread. Additionally, if the process terminated, the debugger may no longe have access to the code and cannot decode the program flow of this process.

On ARM architectures, QNX serves the ContextID register with the address space ID (ASID) of the process. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide performance information of threads.

To allow a detailed performance analysis on QNX threads, the context ID must contain the thread ID. See **Task Runtime Analysis** for an appropriate patch.

## QNX specific Menu for Tracing

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

# Debugging QNX Components

QNX runs on virtual address spaces. The kernel uses a static address translation. Each user process gets its own user address space when loaded, mapped to any physical RAM area that is currently free. Due to this address translations, debugging the QNX kernel and the user processes requires some settings to the debugger.

To distinguish those different memory mappings, TRACE32 uses “space IDs”, defining individual address translations for each ID. The kernel itself (procnto) is attached to the space ID zero. Each process that has its own memory space, gets a space ID that corresponds (but is not equal to) its process ID. QNX threads get the space ID of the parent process.

See also chapter “**MMU Support**”.

## Initial Program Loader (IPL)

The IPL usually resides in Flash on the target to allow reloading of the system image via any target interface. If you’re using an IPL, and you want to debug it, simply load the symbols of the appropriate IPL image into the debugger. The image is located in the QNX SDK directory target/qnx6/<*arch*>/boot/sys.

Example:

```
Data.LOAD.Elf ipl-sengine /NoCODE
```

## QNX Kernel

The QNX system builder generates an image (IFS) that contains the startup code, the kernel and any given application. The file format of the IFS depends on the target system, usually it is in ELF or binary format.

Additionally, the QNX Awareness needs the symbols of the procnto kernel. Please see section “**Hooks & Internals**” how to generate the symbol files of the kernel.

## Downloading the QNX Image

If you start the QNX image from Flash, or if you download the image using the IPL, do this as you are doing it without debugging.

If you want to download the QNX image using the debugger, you have to watch about the file IFS format. If the IFS is in ELF format, simply download this to the target. If the IFS is in binary format, you have to tell the debugger at which address to download it. Please check the example scripts, which version to use and how to obtain the download address.

**Examples**:

```
Data.Load.Elf mbx800.ifs                           ; downloading ELF IFS

Data.Load.Binary pxa250tmdp.ifs 0xa0800000         ; downloading binary IFS
```

To create the IFS in ELF format (in QNX 6.3 and up), change in the System Builder Project (project.bld) the property "System -> Boot File" to "elf".

When downloading the kernel via the debugger, remember to set startup parameters that the kernel requires, before booting the kernel. Usually the IPL passes these parameters to the image.

## Debugging the Kernel Startup

The kernel image starts with a special startup routine, called "startup-*<board>*". If you want to debug this (tiny) startup sequence, you have to load the symbols of this module. If you generated the procnto symbol file, the system builder also preserved the symbol file of the startup image.

**Example**:

```
Data.LOAD.ELF startup-pxa250tmdp.sym /NoCODE
```

As soon as the startup sequence ended, you have to load the kernel symbols. See the next chapter on how to debug the kernel in the virtual address space.

## Debugging the Kernel

For debugging the kernel itself, and for using the QNX awareness, you have to load the virtual addressed symbols of the kernel into the debugger. The procnto symbol file contains all addresses in virtual format, so it's enough to simply load the file:

```
Data.Load.Elf procnto.sym /NoCODE
```

## User Processes

Each user process in QNX gets its own virtual memory space. To distinguish the different memory spaces, the debugger assigns a "space ID", which correlates (but is not equal) to the process ID. Using this space ID, it is possible to address a unique memory location, even if several processes use the same virtual address.

Note that at every time the QNX awareness is used, it needs the kernel symbols. Please see the chapters above on how to load them. Hence, load all process symbols with the option `/NoClear` to preserve the kernel symbols.

| | |
|---|---|
| **NOTE:** | **Debug Builds:** By default, the QNX IDE builds two binaries for the process, one with optimization (e.g. "hello"), and one with debug information, usually with the suffix "_g" (e.g. "hello_g"). Those files contain different code, do not mix them! To be able to debug the process, use the debug variant on *both* sides, i.e. start "hello_g" on the target system, and load the symbol file "hello_g" into the debugger. |

| | |
|---|---|
| **NOTE:** | **Regarding ASLR:** If you use address space layout randomization (ASLR) with "position independent executable" (PIE) code, then use the symbol autoloader to load the symbols of processes and libraries. Each time you invoke a process or a library, it will be loaded onto a different address, making it almost impossible to load the symbols manually. The symbol autoloader takes care of the dynamic loading and loads the symbols to the appropriate locations. |

To correlate the symbols of a user process with the virtual addresses of this process, it is necessary to load the symbols into this space ID.

**Manually Load Process Symbols:**

For example, if you've got a a process called "hello" with the space ID 12 (the dot specifies a decimal number!):

```
Data.LOAD.Elf hello 12.:0 /NoCODE /NoClear
```

The space ID of a process may also be calculated by using the PRACTICE function `task.proc.spaceid()` (see chapter "**QNX PRACTICE Functions**").

**Automatically Load Process Symbols:**

If a process name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command

```
TASK.sYmbol.LOAD "hello"                ; load symbols of "hello"
```

This command loads the symbols of "hello". See **TASK.sYmbol.LOAD** for more information.

**Using the Symbol Autoloader:**

If the symbol autoloader is configured (see chapter "**Symbol Autoloader**"), the symbols will be automatically loaded when accessing an address inside the process. You can also force the loading of the symbols of a process with

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "hello"
```

**Using the Menus:**

Select the menu item "QNX" -> "Process Debugging" -> "Load Symbols" to load the symbols of a specific process. Alternatively, select "Display Processes", right click on the "magic" of a process, and select "Load Symbols".

**Debugging a Process From Scratch, Using a Script:**

If you want to debug your process right from the beginning (at “main()”), you have to load the symbols *before* starting the process. This is a tricky thing because you have to know the process ID, which is assigned first at the process start-up. The demo directory contains a script “app_debug.cmm” that assists you for this purpose. Call the script with the process name as argument before the process is started:

```
DO ~~/demo/<cpu>/kernel/qnx/app_debug.cmm hello
```

Then, start the process in QNX. The debugger should automatically halt at the entry point of the process. You can also use the menu item “QNX” -> “Process Debugging” -> “Debug Process on main...”, which does essentially the same within a dialog. See also chapter “**QNX Specific Menu**”.

**Debugging a Process From Scratch, with Automatic Detection:**

The **TASK.Watch** command group implements the above script as an automatic handler and keeps track of a process launch and the availability of the process symbols. See **TASK.Watch.View** for details.

## Debugging into Shared Libraries

If the process uses shared libraries, QNX loads them into the address space of the process. The process itself contains no symbols of the libraries. If you want to debug those libraries, you have to load the corresponding symbols into the debugger.

**Manually Load Library Symbols:**

1. Start your process and open a **TASK.Process** window.
2. Double-click the magic value of the process that uses the library.
3. Expand the “libraries” tree (if available).

   A list will appear that shows the loaded libraries and the corresponding load addresses.
4. Load the symbols to this address and into the space ID of the process.

   E.g. if the process has the space ID 12., the library is called “lib.so.2” and it is loaded on address 0x01000000, then use the command:

```
Data.LOAD.Elf lib.so.2 12.:0x01000000 /NoCODE /NoClear
```

You can also use PRACTICE functions to automatically load the symbols of a library with a script. E.g.:

```
local &spaceid &magic &address

&spaceid=task.proc.space("hello")
&magic=task.proc.magic("hello")
&address=task.lib.address("lib.so.2",&magic)

Data.LOAD.Elf mylib &spaceid:&address /NoCODE /NoClear
```

Of course, this library must be compiled with debugging information.

**Automatically Load Library Symbols:**

If a library name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command:

```
TASK.sYmbol.LOADLib "hello" "libc.so.2"
```

This command loads the symbols of the library “libc.so”, used by the process “hello”. See **TASK.sYmbol.LOADLib** for more information.

**Using the Symbol Autoloader:**

If the symbol autoloader is configured (see chapter “**Symbol Autoloader**”), the symbols will be automatically loaded when accessing an address inside the library. You can also force the loading of the symbols of a library with:

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "libc.so.2"
```

**Using the Menus:**

Select the menu item “QNX” -> “LIbrary Debugging” -> “Load Symbols” to load the symbols of a specific library. Alternatively, select “Display Processes”, double click on the “magic” of the process, expand the “libraries” section, right click on the “magic” of a library and select “Load Symbols”.

## Debugging QNX Threads

QNX threads share the same virtual memory of the parent process. The OS Awareness for QNX assigns one space ID for all threads that belong to a specific process. It is sufficient, to load the debug information of this process only once (onto its space ID) to debug all threads of this process. See chapter “**Debugging the Process**” for loading the process’ symbols.

The **TASK.Thread** window shows which thread is currently running (“running”).

## Trapping Segmentation Violation

“Segmentation Violation” happens, if the code tries to access a memory location that cannot be mapped in an appropriate way. E.g. if a process tries to write to a read-only area, or if the kernel tries to read from a non-existent address. A user segmentation violation is detected inside the kernel routine “usr_fault()”, if the mapping of page fails.

To trap segmentation violations, set a breakpoint onto the label “usr_fault”. This function is called with three parameters:

- “code_signo” that specifies the signal codes delivered to the thread;
- “thread” specifies, which thread caused the fault;
- “fault_addr” is the address that caused the fault.

On ARM systems these parameters are stored in R0, R1 and R2.
On PowerPC systems these parameters are stored in R3, R4 and R5.

When halted at “usr_fault”, you may load the temporary register set of TRACE32 with the values that are stored in the thread structure of the faulting thread. See the example script “segv.cmm” in the ~~/demo directory.

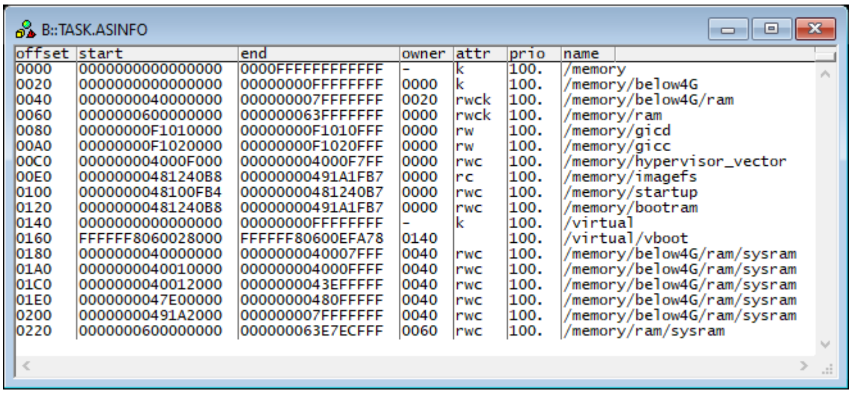
Use **Data.List**, **Var.Local** etc. then to analyze the fault.

As soon as debugging is continued (e.g. **Step**, **Go**, ...), the original register settings at “bad_area” are restored.

## TASK.ASINFO — Display address space information

Format: **TASK.ASINFO**

Displays information about the QNX address spaces, similar to the “pidin syspage=asinfo” command of QNX. This command is available, even if “pidin” is not included in your image.



```
offset start            end              owner attr prio name
0000   0000000000000000 0000FFFFFFFFFFFF -     k    100. /memory
0020   0000000000000000 00000000FFFFFFFF 0000  k    100. /memory/below4G
0040   0000000040000000 000000007FFFFFFF 0020  rwck 100. /memory/below4G/ram
0060   0000000600000000 000000063FFFFFFF 0000  rwck 100. /memory/ram
0080   00000000F1010000 00000000F1010FFF 0000  rw   100. /memory/gicd
00A0   00000000F1020000 00000000F1020FFF 0000  rw   100. /memory/gicc
00C0   000000004000F000 000000004000F7FF 0000  rwc  100. /memory/hypervisor_vector
00E0   00000000481240B8 00000000491A1FB7 0000  rc   100. /memory/imagefs
0100   0000000048100FB4 00000000481240B7 0000  rwc  100. /memory/startup
0120   00000000481240B8 00000000491A1FB7 0000  rwc  100. /memory/bootram
0140   0000000000000000 00000000FFFFFFFF -     k    100. /virtual
0160   FFFFFF8060028000 FFFFFF80600EFA78 0140       100. /virtual/vboot
0180   0000000040000000 0000000040007FFF 0040  rwc  100. /memory/below4G/ram/sysram
01A0   0000000040010000 000000004000FFFF 0040  rwc  100. /memory/below4G/ram/sysram
01C0   0000000040012000 0000000043EFFFFF 0040  rwc  100. /memory/below4G/ram/sysram
01E0   0000000047E00000 00000000480FFFFF 0040  rwc  100. /memory/below4G/ram/sysram
0200   00000000491A2000 000000007FFFFFFF 0040  rwc  100. /memory/below4G/ram/sysram
0220   0000000600000000 000000063E7ECFFF 0060  rwc  100. /memory/ram/sysram
```

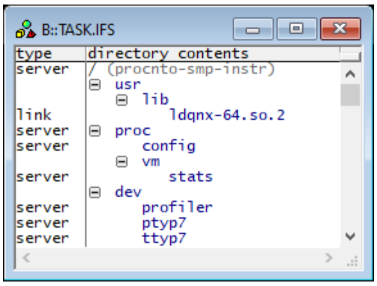
## TASK.IFS — Display directory of IFS

Format: **TASK.IFS** [*<process>*]

Displays the directory contents of the Image File System (IFS).

This command shows only the directories and files that are available within the IFS. It does not show directories and files on a different file system, even if it is linked into the IFS.

| | |
|---|---|
| *<process>* | Specify a process magic or name to show the root file system of this process.<br><br>If left empty, the root file system of the kernel is shown. |



```
type    directory contents
server  / (procnto-smp-instr)
          usr
            lib
link          ldqnx-64.so.2
server    proc
server      config
            vm
server        stats
          dev
server      profiler
server      ptyp7
server      ttyp7
```

## TASK.MMU.SCAN

Scan process MMU space

Format: **TASK.MMU.SCAN** [*<process>*]

Scans the target MMU of the space ID, specified by the given process, and sets the Debugger MMU appropriately, to cover the physical to logical address translation of this specific process.

The command walks through all page tables which are defined for the memory spaces of the process and prepares the Debugger MMU to hold the physical to logical address translation of this process. This is needed to provide full HLL support. If a process was loaded dynamically, you must set the Debugger MMU to this process, otherwise the Debugger won't know where the physical image of the process is placed.

To successfully execute this command, space IDs must be enabled (**SYStem.Option.MMUSPACES ON**).

| | |
|---|---|
| *<process>* | Specify a process magic, space ID or name.<br>If no argument is specified, the command scans all current processes. |

**Example**:

```
; scan the memory space of the process "hello"
TASK.MMU.SCAN "hello"
```

See also **MMU Support**.

## TASK.Option

Set awareness options

| | |
|---|---|
| Format: | **TASK.Option** *<option>* |
| *<option>*: | **THRCTX** [**ON** \| **OFF**] |
| | **TTBHV** *<address>* |

Set various options to the awareness.

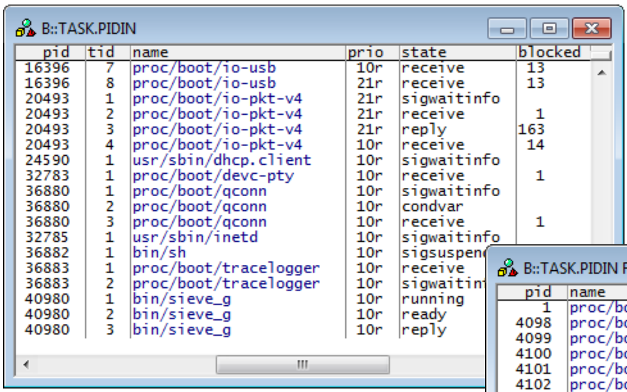
| | |
|---|---|
| **THRCTX** | Set the context ID type that is recorded with the real-time trace (e.g. ETM). If set to on, the context ID in the trace contains thread switch detection. See **Task Runtime Statistics**. |
| **TTBHV** | If QNX is used as a hypervisor, this command sets the translation table base address of the hypervisor. This is necessary to allow the awareness access to the hypervisor internals, even if currently a guest is active. |

## TASK.PIDIN

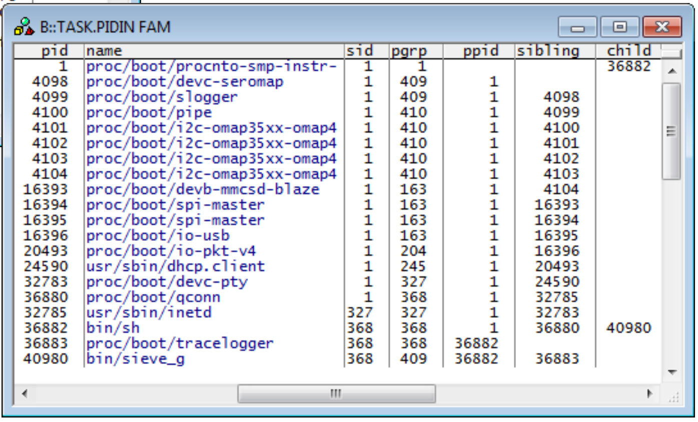
Display “pidin” like information

| | |
|---|---|
| Format: | **TASK.PIDIN** [**FAm** \| **FLags** \| **PMEM** \| **MEM**] |

Displays information like the “pidin” command of QNX *without* using “pidin” itself or any other kernel resources. This command is available, even if “pidin” is not included in your image.



```
  pid  tid  name                  prio  state        blocked
16396    7  proc/boot/io-usb       10r  receive       13
16396    8  proc/boot/io-usb       21r  receive       13
20493    1  proc/boot/io-pkt-v4    21r  sigwaitinfo
20493    2  proc/boot/io-pkt-v4    21r  receive        1
20493    3  proc/boot/io-pkt-v4    21r  reply        163
20493    4  proc/boot/io-pkt-v4    10r  receive       14
24590    1  usr/sbin/dhcp.client   10r  sigwaitinfo
32783    1  proc/boot/devc-pty     10r  receive        1
36880    1  proc/boot/qconn        10r  sigwaitinfo
36880    2  proc/boot/qconn        10r  condvar
36880    3  proc/boot/qconn        10r  receive        1
32785    1  usr/sbin/inetd         10r  sigwaitinfo
36882    1  bin/sh                 10r  sigsuspen
36883    1  proc/boot/tracelogger  10r  receive
36883    2  proc/boot/tracelogger  10r  sigwaitin
40980    1  bin/sieve_g            10r  running
40980    2  bin/sieve_g            10r  ready
40980    3  bin/sieve_g            10r  reply
```

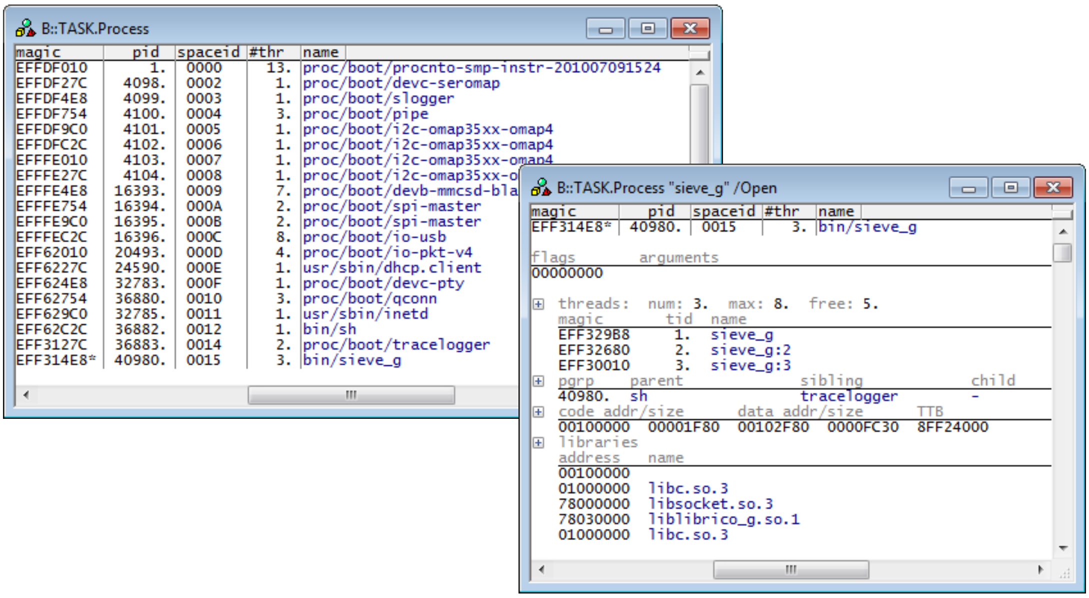


```
  pid  name                            sid  pgrp   ppid  sibling  child
    1  proc/boot/procnto-smp-instr-      1     1                  36882
 4098  proc/boot/devc-seromap            1   409      1
 4099  proc/boot/slogger                 1   409      1     4098
 4100  proc/boot/pipe                    1   410      1     4099
 4101  proc/boot/i2c-omap35xx-omap4      1   410      1     4100
 4102  proc/boot/i2c-omap35xx-omap4      1   410      1     4101
 4103  proc/boot/i2c-omap35xx-omap4      1   410      1     4102
 4104  proc/boot/i2c-omap35xx-omap4      1   410      1     4103
16393  proc/boot/devb-mmcsd-blaze        1   163      1     4104
16394  proc/boot/spi-master              1   163      1    16393
16395  proc/boot/spi-master              1   163      1    16394
16396  proc/boot/io-usb                  1   163      1    16395
20493  proc/boot/io-pkt-v4               1   204      1    16396
24590  usr/sbin/dhcp.client              1   245      1    20493
32783  proc/boot/devc-pty                1   327      1    24590
36880  proc/boot/qconn                   1   368      1    32785
32785  usr/sbin/inetd                  327   327      1    32783
36882  bin/sh                          368   368      1    36880  40980
36883  proc/boot/tracelogger           368   368  36882
40980  bin/sieve_g                     368   409  36882    36883
```

Format: **TASK.Process** [*<process>*]

Displays the process table of QNX or detailed information about one specific process.

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



```
magic         pid  spaceid #thr  name
EFFDF010       1.   0000    13.  proc/boot/procnto-smp-instr-201007091524
EFFDF27C    4098.   0002     1.  proc/boot/devc-seromap
EFFDF4E8    4099.   0003     1.  proc/boot/slogger
EFFDF754    4100.   0004     3.  proc/boot/pipe
EFFDF9C0    4101.   0005     1.  proc/boot/i2c-omap35xx-omap4
EFFDFC2C    4102.   0006     1.  proc/boot/i2c-omap35xx-omap4
EFFFE010    4103.   0007     1.  proc/boot/i2c-omap35xx-omap4
EFFFE27C    4104.   0008     1.  proc/boot/i2c-omap35xx-o
EFFFE4E8   16393.   0009     7.  proc/boot/devb-mmcsd-bla
EFFFE754   16394.   000A     2.  proc/boot/spi-master
EFFFE9C0   16395.   000B     2.  proc/boot/spi-master
EFFFEC2C   16396.   000C     8.  proc/boot/io-usb
EFF62010   20493.   000D     4.  proc/boot/io-pkt-v4
EFF6227C   24590.   000E     1.  usr/sbin/dhcp.client
EFF624E8   32783.   000F     1.  proc/boot/devc-pty
EFF62754   36880.   0010     3.  proc/boot/qconn
EFF629C0   32785.   0011     1.  usr/sbin/inetd
EFF62C2C   36882.   0012     1.  bin/sh
EFF3127C   36883.   0014     2.  proc/boot/tracelogger
EFF314E8*  40980.   0015     3.  bin/sieve_g
```

```
magic         pid  spaceid #thr  name
EFF314E8*  40980.   0015     3.  bin/sieve_g

flags       arguments
00000000

⊞ threads:  num: 3.  max: 8.  free: 5.
  magic        tid  name
  EFF329B8      1.  sieve_g
  EFF32680      2.  sieve_g:2
  EFF30010      3.  sieve_g:3
⊞ pgrp    parent              sibling            child
  40980.  sh                  tracelogger        -
⊞ code addr/size       data addr/size      TTB
  00100000  00001F80   00102F80  0000FC30  8FF24000
⊞ libraries
  address   name
  00100000
  01000000  libc.so.3
  78000000  libsocket.so.3
  78030000  liblibrico_g.so.1
  01000000  libc.so.3
```

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

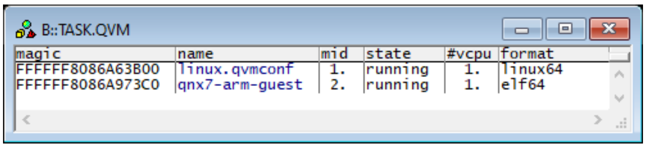
The fields “magic”, “parent”, “sibling” and “child” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on them will show a local menu.

## TASK.QVM Display VMs

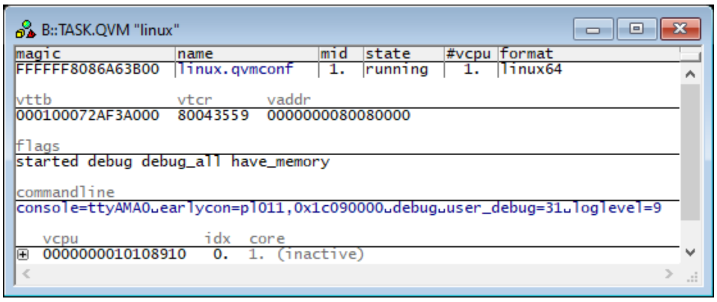
| Format: | **TASK.QVM** [*<vm>*] |
|---|---|

If QNX is used as a hypervisor, this command displays a table of VMs or detailed information about one specific VM.

Without any arguments, a table with all created VMs will be shown. Specify a VM magic, name or ID to display detailed information about that VM.



```
magic            name             mid  state    #vcpu format
FFFFFF8086A63B00 linux.qvmconf     1.  running    1.  linux64
FFFFFF8086A973C0 qnx7-arm-guest    2.  running    1.  elf64
```



```
magic            name             mid  state    #vcpu format
FFFFFF8086A63B00 linux.qvmconf     1.  running    1.  linux64

vttb             vtcr     vaddr
000100072AF3A000 80043559 0000000080080000

flags
started debug debug_all have_memory

commandline
console=ttyAMA0␣earlycon=pl011,0x1c090000␣debug␣user_debug=31␣loglevel=9

    vcpu             idx  core
⊞   0000000010108910  0.  1. (inactive)
```

| **NOTE:** | This feature is only available if at least one qvm process is running, and if the symbol information of the process "qvm" is loaded. |
|---|---|

## TASK.SHMEM Display contents of shmem

| Format: | **TASK.SHMEM** *<shmem_file_path>* |
|---|---|

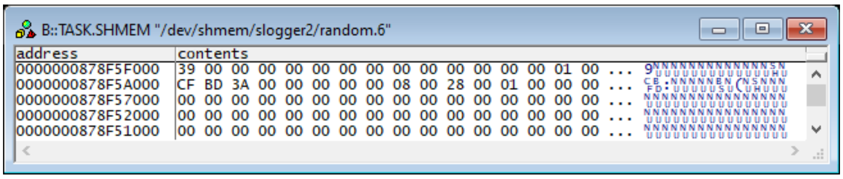
Displays the contents of the physical address pages of the given shared memory file.

| | |
|---|---|
| *<shmem_file_path>* | Specify a fully qualified path name to the shmem file. |

**Example**:

```
; show the contents of shared memory named “myshmem”

TASK.SHMEM “/dev/shmem/myshmem”
```



B::TASK.SHMEM "/dev/shmem/slogger2/random.6"

```
address          contents
0000000878F5F000 39 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 ...
0000000878F5A000 CF BD 3A 00 00 00 00 00 08 00 28 00 01 00 00 00 ...
0000000878F57000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0000000878F52000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0000000878F51000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

## TASK.SLOGGER2 Display contents of slogger2 buffers

Format: **TASK.SLOGGER2** *<slogger2_buffer_name>*

Displays the contents of buffers created by slogger2.

| | |
|---|---|
| *<slogger2_buffer_name>* | Specify a slogger 2 buffer name. |

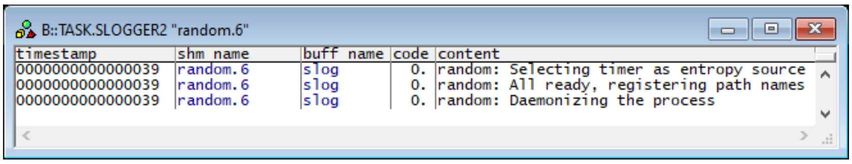
**NOTE:** This feature heavily depends on the used QNX version and slogger2 daemon.

Contact Lauterbach support if you see inconsistencies in the buffer display.

**Example**:

```
; show the contents of slogger2 buffer named "random.6"

TASK.SLOGGER2 "random.6"
```



B::TASK.SLOGGER2 "random.6"

```
timestamp        shm name     buff name code content
0000000000000039 random.6     slog         0. random: Selecting timer as entropy source
0000000000000039 random.6     slog         0. random: All ready, registering path names
0000000000000039 random.6     slog         0. random: Daemonizing the process
```

## TASK.sYmbol                                        Process symbol management

The **TASK.sYmbol** command group helps to load and unload symbols of a given process or library. In particular the commands are:

| | |
|---|---|
| **TASK.sYmbol.LOAD** | Load process symbols and MMU |
| **TASK.sYmbol.LOAD** | Unload process symbols and MMU |
| **TASK.sYmbol.LOADLib** | Load library symbols |
| **TASK.sYmbol.DELeteLib** | Unload library symbols |
| **TASK.sYmbol.Option** | Set symbol management options |

## TASK.sYmbol.DELete                                Unload process symbols and MMU

| | |
|---|---|
| Format: | **TASK.sYmbol.DELete** *<process>* |

When debugging of a process is finished, or if the process exited, you should remove loaded process symbols and MMU entries. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified process.

| | |
|---|---|
| *<process>* | Specify the process name (in quotes) or magic to unload the symbols of this process. |

## TASK.sYmbol.DELeteLib — Unload library symbols

| | |
|---|---|
| Format: | **TASK.sYmbol.DELeteLib** *<process> <library>* |

When debugging of a library is finished, or if the library is removed from the kernel, you should remove loaded library symbols. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified library.

| | |
|---|---|
| *<process>* | Specify the process to which the desired library belongs (name in quotes or magic). |
| *<library>* | Specify the library name in quotes. The library name **must** match the name as shown in **TASK.Process** *<process>*, ”libraries”. |

**Example**:

```
TASK.sYmbol.DELeteLib “hello" "libc-2.2.1.so"
```

See also chapter “**Debugging Into Shared Libraries**”.

## TASK.sYmbol.LOAD — Load process symbols and MMU

| | |
|---|---|
| Format: | **TASK.sYmbol.LOAD** *<process>* |

Specify the process name (in quotes) or magic to load the symbols of this process.

In order to debug a user process, the debugger needs the symbols of this process (see chapter “Debugging User Processes”).
This command retrieves the appropriate space ID and loads the symbol file of an existing process. Note that this command works only with processes that are already loaded in QNX (i.e. processes that show up in the **TASK.Process** window).

The actual command used for loading the symbols can be changed with **TASK.sYmbol.Option LOADCMD**.

## TASK.sYmbol.LOADLib — Load library symbols

| | |
|---|---|
| Format: | **TASK.sYmbol.LOADLib** *<process>* *<library>* |

As first parameter, specify the process to which the desired library belongs (name in quotes or magic). Specify the library name in quotes as second parameter. The library name **must** match the name as shown in **TASK.Process** *<process>*, ”libraries”.

In order to debug a library, the debugger needs the symbols of this library, relocated to the correct addresses where QNX linked this library. This command retrieves the appropriate load addresses and loads the .so symbol file of an existing library. Note that this command works only with libraries that are already loaded in QNX (i.e. libraries that show up in the **TASK.Process** *<process>* window).

Example:

```
TASK.sYmbol.LOADLib "hello" "libc-2.2.1.so"
```

See also chapter “**Debugging Into Shared Libraries**”.

## TASK.sYmbol.Option — Set symbol management options

| | |
|---|---|
| Format: | **TASK.sYmbol.Option** *<option>* |
| *<option>*: | **LOADCMD** *<command>*<br>**LOADLCMD** *<command>*<br>**MMUSCAN** [**ON** \| **OFF**]<br>**AutoLoad** *<option>* |

Set a specific option to the symbol management.

**LOADCMD:**

This setting is only active, if the symbol autoloader for processes is off.

**TASK.sYmbol.LOAD** uses a default load command to load the symbol file of the process. This loading command can be customized using this option with the command enclosed in quotes. Two parameters are passed to the command in a fixed order:

| | |
|---|---|
| %s | Name of the process |
| %x | Space ID of the process |

Examples:

```
TASK.sYmbol.Option LOADCMD "Data.LOAD.Elf %s 0x%x:0 /NoCODE /NoClear"

TASK.sYmbol.Option LOADCMD "DO myloadscript %s 0x%x"
```

**LOADLCMD:**

This setting is only active, if the symbol autoloader for libraries is off.

**TASK.sYmbol.LOADLib** uses a default load command to load the symbol file of the library. This loading command can be customized using this option with the command enclosed in quotes. Three parameters are passed to the command in a fixed order:

| | |
|---|---|
| %s | name of the library |
| %x | space ID of the library |
| %x | load address of the library. |

Examples:

```
 TASK.sYmbol.Option LOADLCMD "D.LOAD.Elf %s 0x%x:0x%x /NoCODE /NoClear"

 TASK.sYmbol.Option LOADLCMD "DO myloadlscript %s 0x%x 0x%x"
```

**MMUSCAN:**

This option controls, if the symbol loading mechanisms of **TASK.sYmbol** scan the MMU page tables of the loaded components, too. When using **TRANSlation.TableWalk**, then switch this off.

**AutoLoad:**

This option controls, which components are checked and managed by the **symbol autoloader**:

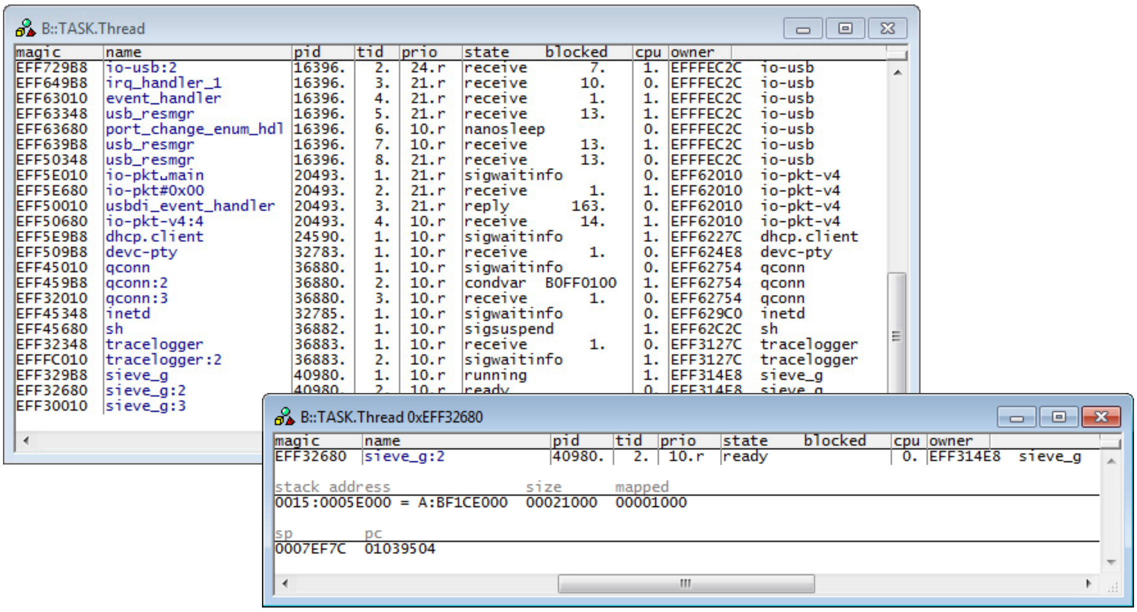
| | |
|---|---|
| **Process** | Check processes |
| **Library** | Check all libraries of all processes |
| **CurrLib** | Check only libraries of current process |
| **ALL** | Check processes, and all libraries |
| **NoProcess** | Don’t check processes |
| **NoLibrary** | Don’t check libraries |
| **NONE** | Check nothing. |

The options are set *additionally*, not removing previous settings.

| Format: | **TASK.Thread** [<*thread*>] |
|---|---|

Displays the thread table of QNX or detailed information about one specific thread.

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



```
magic    name                pid     tid  prio  state      blocked   cpu owner
EFF729B8 io-usb:2            16396.   2.  24.r  receive         7.    1. EFFFEC2C  io-usb
EFF649B8 irq_handler_1       16396.   3.  21.r  receive        10.    0. EFFFEC2C  io-usb
EFF63010 event_handler       16396.   4.  21.r  receive         1.    1. EFFFEC2C  io-usb
EFF63348 usb_resmgr          16396.   5.  21.r  receive        13.    1. EFFFEC2C  io-usb
EFF63680 port_change_enum_hdl 16396.  6.  10.r  nanosleep             0. EFFFEC2C  io-usb
EFF639B8 usb_resmgr          16396.   7.  10.r  receive        13.    1. EFFFEC2C  io-usb
EFF50348 usb_resmgr          16396.   8.  21.r  receive        13.    0. EFFFEC2C  io-usb
EFF5E010 io-pkt␣main         20493.   1.  21.r  sigwaitinfo           0. EFF62010  io-pkt-v4
EFF5E680 io-pkt#0x00         20493.   2.  21.r  receive         1.    1. EFF62010  io-pkt-v4
EFF50010 usbdi_event_handler 20493.   3.  21.r  reply         163.    0. EFF62010  io-pkt-v4
EFF50680 io-pkt-v4:4         20493.   4.  10.r  receive        14.    1. EFF62010  io-pkt-v4
EFF5E9B8 dhcp.client         24590.   1.  10.r  sigwaitinfo           1. EFF6227C  dhcp.client
EFF509B8 devc-pty            32783.   1.  10.r  receive         1.    0. EFF624E8  devc-pty
EFF45010 qconn               36880.   1.  10.r  sigwaitinfo           0. EFF62754  qconn
EFF459B8 qconn:2             36880.   2.  10.r  condvar   B0FF0100    1. EFF62754  qconn
EFF32010 qconn:3             36880.   3.  10.r  receive         1.    0. EFF62754  qconn
EFF45348 inetd               32785.   1.  10.r  sigwaitinfo           0. EFF629C0  inetd
EFF45680 sh                  36882.   1.  10.r  sigsuspend            1. EFF62C2C  sh
EFF32348 tracelogger         36883.   1.  10.r  receive         1.    0. EFF3127C  tracelogger
EFFFC010 tracelogger:2       36883.   2.  10.r  sigwaitinfo           1. EFF3127C  tracelogger
EFF329B8 sieve_g             40980.   1.  10.r  running               1. EFF314E8  sieve_g
EFF32680 sieve_g:2           40980.   2.  10.r  ready                 0. EFF314E8  sieve_g
EFF30010 sieve_g:3
```

```
magic    name                pid     tid  prio  state      blocked   cpu owner
EFF32680 sieve_g:2           40980.   2.  10.r  ready                 0. EFF314E8  sieve_g

stack address                size      mapped
0015:0005E000 = A:BF1CE000   00021000  00001000

sp        pc
0007EF7C  01039504
```

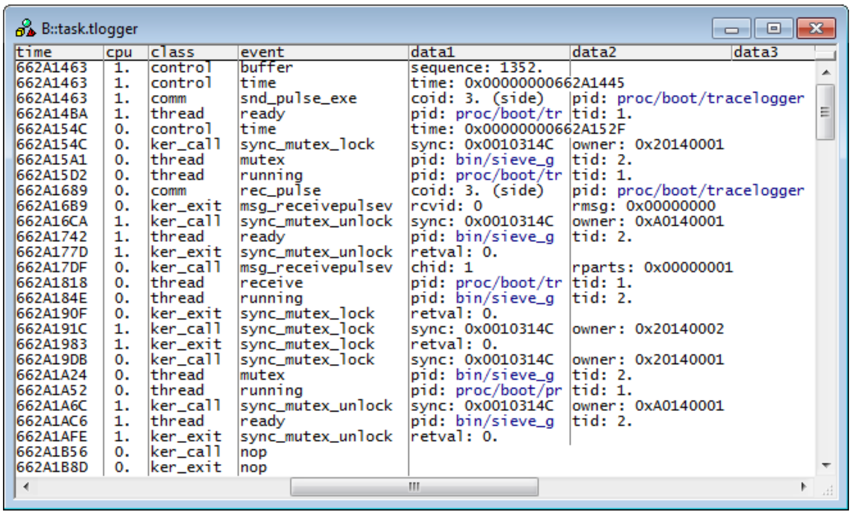
"magic" is a unique ID, used by the OS Awareness to identify a specific thread (address of the TCB).

The fields "magic" and "owner" are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

Format: **TASK.TLOGger** [*<option>* [**/***<option>* [**...**]]]

*<option>*: **Reverse**
**Filter Control** | **Kercall** | **Int** | **Process** | **Thread** | **coMm**

**TASK.TLOGger** displays the kernel internal buffer of the kernel tracelogger feature.



```
time      cpu  class     event             data1                data2                       data3
662A1463  1.   control   buffer            sequence: 1352.
662A1463  1.   control   time              time: 0x00000000662A1445
662A1463  1.   comm      snd_pulse_exe     coid: 3. (side)      pid: proc/boot/tracelogger
662A14BA  1.   thread    ready             pid: proc/boot/tr    tid: 1.
662A154C  0.   control   time              time: 0x00000000662A152F
662A154C  0.   ker_call  sync_mutex_lock   sync: 0x0010314C     owner: 0x20140001
662A15A1  0.   thread    mutex             pid: bin/sieve_g     tid: 2.
662A15D2  0.   thread    running           pid: proc/boot/tr    tid: 1.
662A1689  0.   comm      rec_pulse         coid: 3. (side)      pid: proc/boot/tracelogger
662A16B9  0.   ker_exit  msg_receivepulsev rcvid: 0             rmsg: 0x00000000
662A16CA  1.   ker_call  sync_mutex_unlock sync: 0x0010314C     owner: 0xA0140001
662A1742  1.   thread    ready             pid: bin/sieve_g     tid: 2.
662A177D  1.   ker_exit  sync_mutex_unlock retval: 0.
662A17DF  0.   ker_call  msg_receivepulsev chid: 1              rparts: 0x00000001
662A1818  0.   thread    receive           pid: proc/boot/tr    tid: 1.
662A184E  0.   thread    running           pid: bin/sieve_g     tid: 2.
662A190F  0.   ker_exit  sync_mutex_lock   retval: 0.
662A191C  1.   ker_call  sync_mutex_lock   sync: 0x0010314C     owner: 0x20140002
662A1983  1.   ker_exit  sync_mutex_lock   retval: 0.
662A19DB  0.   ker_call  sync_mutex_lock   sync: 0x0010314C     owner: 0x20140001
662A1A24  0.   thread    mutex             pid: bin/sieve_g     tid: 2.
662A1A52  0.   thread    running           pid: proc/boot/pr    tid: 1.
662A1A6C  1.   ker_call  sync_mutex_unlock sync: 0x0010314C     owner: 0xA0140001
662A1AC6  1.   thread    ready             pid: bin/sieve_g     tid: 2.
662A1AFE  1.   ker_exit  sync_mutex_unlock retval: 0.
662A1B56  0.   ker_call  nop
662A1B8D  0.   ker_exit  nop
```

See QNX documentation for tracelogger. “tracelogger” is only available in intrumented QNX kernels and must be started in QNX to fill the kernel buffers

**TASK.TLOGger** *only* displays the kernel buffers. As soon as they are flushed to the file, they're gone from the kernel buffers. I.e. **TASK.TLOGger** shows only data as long as "tracelogger" is still active.

| | |
|---|---|
| **Reverse** | Displays the most recent entries first. |
| **Filter** | Filter the given class of event. You can specify this option several times to filter several classes. |

**Example**:

```
; display tracelogger buffer in reverse order and
; do not display kernel call events and control events
TASK.TLOGger /Reverse /Filter Kercall /Filter Control
```

Format: **TASK.TLOGger.VMLOGger** [*<cpu>*]

**TASK.TLOGger.VMLOGger** copies the thread state entries of the kernel internal tracelogger buffer to a debugger-internal buffer in virtual memory (VM:), using the **LOGGER** structure layout.
In SMP systems, specify the cpu number of the events to copy.

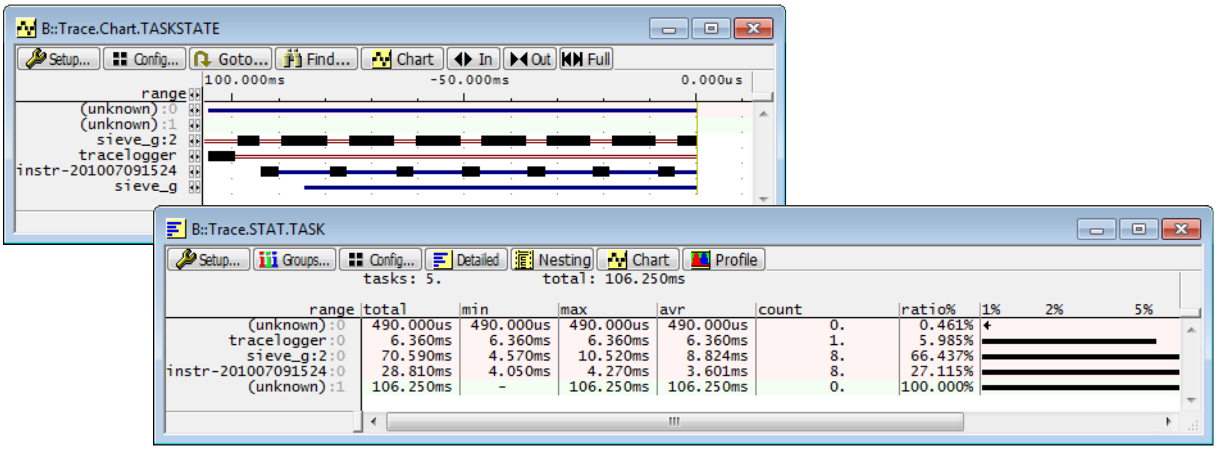
See QNX documentation for tracelogger. “tracelogger” is only available in intrumented QNX kernels and must be started in QNX to fill the kernel buffers

**TASK.TLOGger.VMLOGger** *only* copies the kernel buffers. As soon as they are flushed to the file, they're gone from the kernel buffers. I.e. **TASK.TLOGger.VMLOGger** works only as long as "tracelogger" is still active.

Activate the LOGGER and copy the buffers with:

```
Trace.METHOD Logger
Logger.RESet
Logger.ADDRESS AVM:0x1000
Logger.TimeStamp Up
Logger.TimeStamp.Rate 100000.
Logger.Init
TASK.TLOGger.VMLOGger
Logger.ARM
Logger.OFF
```

After this, you can use the Logger contents for **Task Runtime Statistics** and **Task State Analysis**.



```
                  tasks: 5.          total: 106.250ms

              range |total     |min       |max       |avr       |count          |ratio%  |1%     2%        5%
       (unknown):0  | 490.000us| 490.000us| 490.000us| 490.000us|            0. |  0.461%|
      tracelogger:0 |   6.360ms|   6.360ms|   6.360ms|   6.360ms|            1. |  5.985%|
        sieve_g:2:0 |  70.590ms|   4.570ms|  10.520ms|   8.824ms|            8. | 66.437%|
instr-201007091524:0|  28.810ms|   4.050ms|   4.270ms|   3.601ms|            8. | 27.115%|
       (unknown):1  | 106.250ms|     -    | 106.250ms| 106.250ms|            0. |100.000%|
```

## TASK.Watch — Watch processes

The TASK.Watch command group builds a watch system that watches your QNX target for specified processes. It loads and unloads process symbols automatically. Additionally it covers process creation and may stop watched processes at their entry points.

In particular the watch commands are:

| | |
|---|---|
| **TASK.Watch.View** | Activate watch system and show watched processes |
| **TASK.Watch.ADD** | Add process to watch list |
| **TASK.Watch.DELete** | Remove process from watch list |
| **TASK.Watch.DISable** | Disable watch system |
| **TASK.Watch.ENable** | Enable watch system |
| **TASK.Watch.DISableBP** | Disable process creation breakpoints |
| **TASK.Watch.ENableBP** | Enable process creation breakpoints |

## TASK.Watch.ADD — Add process to watch list

| | |
|---|---|
| Format: | **TASK.Watch.ADD** *<process>* |

Adds a process to the watch list.

| | |
|---|---|
| *<process>* | Specify the process name (in quotes) or magic. |

Please see **TASK.Watch.View** for details.

## TASK.Watch.DELete — Remove process from watch list

| | |
|---|---|
| Format: | **TASK.Watch.DELete** *<process>* |

Removes a process from the watch list.

| | |
|---|---|
| *<process>* | Specify the process name (in quotes) or magic. |

Please see **TASK.Watch.View** for details.

## TASK.Watch.DISable — Disable watch system

|  |  |
|---|---|
| Format: | **TASK.Watch.DISable** |

Disables the complete watch system. The watched processes list is no longer checked against the target and is not updated. You'll see the **TASK.Watch.View** window grayed out.

This feature is useful if you want to keep process symbols in the debugger, even if the process terminated.

## TASK.Watch.DISableBP — Disable process creation breakpoints

|  |  |
|---|---|
| Format: | **TASK.Watch.DISableBP** |

Prevents the debugger from setting on-chip breakpoints for the detection of process creation. After executing this command, the target will run in real-time. However, the watch system can no longer detect process creation. Automatic loading of process symbols will still work.

This feature is useful if you'd like to use the on-chip breakpoints for other purposes.

Please see **TASK.Watch.View** for details.

## TASK.Watch.ENable — Enable watch system

|  |  |
|---|---|
| Format: | **TASK.Watch.ENable** |

Enables the previously disabled watch system. It enables the automatic loading of process symbols as well as the detection of process creation.

Please see **TASK.Watch.View** for details.

## TASK.Watch.ENableBP — Enable process creation breakpoints

| | |
|---|---|
| Format: | **TASK.Watch.ENable** |

Enables the previously disabled on-chip breakpoints for detection of process creation.

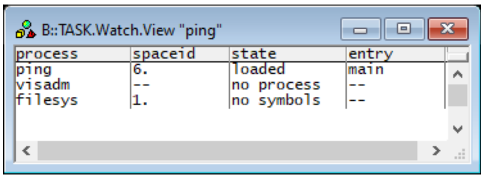
Please see **TASK.Watch.View** for details.

## TASK.Watch.View — Show watched processes

| | |
|---|---|
| Format: | **TASK.Watch.View** [*<process>*] |

Activates the watch system for processes and shows a table of the watched processes.

| | |
|---|---|
| **NOTE:** | **This feature may affect the real-time behavior of the target application!** Please see below for details. |


B::TASK.Watch.View "ping"

```
process   spaceid   state        entry
ping      6.        loaded       main
visadm    --        no process   --
filesys   1.        no symbols   --
```

| | |
|---|---|
| *<process>* | Specify a process name for the initial process to be watched. |

### Description of Columns in the TASK.Watch.View Window

| | |
|---|---|
| **process** | The name of the process to be watched. |
| **spaceid** | The current space ID of the watched process.<br>If grayed, the debugger is currently not able to determine the space ID of the process (e.g. the target is running). |

| | |
|---|---|
| **state** | The current watch state of the process.<br>If grayed, the debugger is currently not able to determine the watch state.<br>**no process**: The debugger couldn't find the process in the current QNX process list.<br>**no symbols**: The debugger found the process and loaded the MMU settings of the process but couldn't load the symbols of the process (most likely because the corresponding symbol files were missing).<br>**loaded**: The debugger found the process and loaded the process's MMU settings and symbols. |
| **entry** | The process entry point, which is `main()`.<br>If grayed, the debugger is currently not able to detect the entry point or is unable to set the process entry breakpoint (e.g. because it is disabled with **TASK.Watch.DISableBP**). |

The watch system for processes is able to automatically load and unload the symbols of a process, depending on their state in the target. Additionally, the watch system can detect the creation of a process and halts the process at its entry point.

| | |
|---|---|
| **TASK.Watch.ADD** | Adds processes to the watch list. |
| **TASK.Watch.DELete** | Removes processes from the watch list. |

The watch system for processes is active as long as the **TASK.Watch.View** window is open or iconized. As soon as this window is closed, the watch system will be deactivated.

### Automatic Loading and Unloading of Process Symbols

In order to detect the current processes, the debugger must have full access to the target, i.e. the target application must be stopped (with one exception, see below for creation of processes). As long as the target runs in real time, the watch system is not able to get the current process list, and the display will be grayed out (inactive).

If the target is halted (either by hitting a breakpoint, or by halting it manually), the watch system starts its work. For each of the processes in the watch list, it determines the state of this process in the target.

If a process is active on the target, which was previously not found there, the watch system loads the appropriate symbol files. In fact, it executes **TASK.sYmbol.LOAD** for the new process.

If a watched process was previously loaded but is no longer found on the QNX process list, the watch system unloads the symbols. The watch system executes **TASK.sYmbol.DELete** for this process.

If the process was previously loaded and is now found with another space ID (e.g. if the process terminated and started again), the watch system first removes the process symbols and reloads them to the appropriate space ID.

You can disable the loading / unloading of process symbols with the command **TASK.Watch.DISable**.

## Detection of Process Creation

To halt a process at its main entry point, the watch system can detect the process creation and set the appropriate breakpoints.

To detect the process creation, the watch system sets an on-chip breakpoint on a kernel function that is called upon creation of processes. Every time the breakpoint is hit, the debugger checks if a watched process is started. If not, it simply resumes the target application. If the debugger detects the start of a newly created (and watched) process, it sets an on-chip breakpoint onto the main entry point of the process (`main()`) and resumes the target application. A short while after this, the main breakpoint will hit and halt the target at the entry point of the process. The process is now ready to be debugged.

| | |
|---|---|
| **NOTE:** | This feature uses one permanent on-chip breakpoint and one temporary on-chip breakpoint when a process is created. Please ensure that at least those two on-chip breakpoints are available when using this feature.<br><br>Upon every process creation, the target application is halted for a short time and resumed after searching for the watched processes. **This impacts the real-time behavior of your target.** |

If you don't want the watch system to set breakpoints, you can disable them with the command **TASK.Watch.DISableBP**. Of course, detection of process creation won't work then.

# QNX PRACTICE Functions

There are special definitions for QNX specific PRACTICE functions.

## TASK.ASINFO.SIZE() Size of address space

Syntax: **TASK.ASINFO.SIZE("***<asinfo_name>***",***<index>***)**

Returns the size of a QNX address space specified by the address space name and index.

**Parameter and Description**:

| | |
|---|---|
| *<asinfo_name>* | **Parameter Type**: String.<br>Name of the QNX address space |
| *<index>* | **Parameter Type**: Decimal or hex or binary value.<br>Index of the entry for the given address space, if an address space covers several ranges.<br>Returns -1 if the index is bigger than available address ranges. |

**Return Value Type**: Hex value.

## TASK.ASINFO.START() Start of address space

Syntax: **TASK.ASINFO.START("***<asinfo_name>***",***<index>***)**

Returns the start address of a QNX address space specified by the address space name and index.

**Parameter and Description**:

| | |
|---|---|
| *<asinfo_name>* | **Parameter Type**: String.<br>Name of the QNX address space |
| *<index>* | **Parameter Type**: Decimal or hex or binary value.<br>Index of the entry for the given address space, if an address space covers several ranges.<br>Returns -1 if the index is bigger than available address ranges. |

**Return Value Type**: Hex value.

## 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.

## TASK.CORE.ASSIGN() Core assignment

x86/x64

Syntax: **TASK.CORE.ASSIGN()**

Returns the core assignment of the specified process.

QNX may change the order of the cores between different runs. This means, core number 1 in QNX may be assigned to different physical cores when rebooting.

This function returns the actual used core assignment string to be used with **CORE.ASSIGN**.

**Return Value Type**: String.

## TASK.CURRENT() Current process or thread

Syntax: **TASK.CURRENT(process | thread | spaceid)**

Return the current process, thread or space ID.

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

**Parameter and Description**:

| | |
|---|---|
| **process** | Returns the current process magic number. |
| **thread** | Returns the current thread magic number. |
| **spaceid** | Returns the current space ID. |

**Return Value Type**: Hex value.

## TASK.LIB.ADDRESS() Address of library

| | |
|---|---|
| Syntax: | **TASK.LIB.ADDRESS("***<library_name>***",***<process_magic>***)** |

Returns the start address of the given library used by the specified process.

**Parameter and Description**:

| | |
|---|---|
| *<library_name>* | **Parameter Type**: String (*with* quotation marks). |
| *<process_magic>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.PROC.ID() Process ID

| | |
|---|---|
| Syntax: | **TASK.PROC.ID(***<process_magic>***)** |

Returns the PID of the specified process.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.PROC.MAGIC() Magic number of process

| | |
|---|---|
| Syntax: | **TASK.PROC.MAGIC("***<process_name>***")** |

Returns the magic number of the specified process.

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

**Return Value Type**: Hex value.

## TASK.PROC.NAME() Name of process

| | |
|---|---|
| Syntax: | **TASK.PROC.NAME(***<process_magic>***)** |

Returns the name of the specified process.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: String.

## TASK.PROC.SID2MAGIC() Process of space ID

| | |
|---|---|
| Syntax: | **TASK.PROC.SID2MAGIC(***<space_id>***)** |

Returns the magic number of a process with the given space ID.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.PROC.SPACE() Space ID of process

Syntax: **TASK.PROC.SPACE("***<process_name>***")**

Returns the debugger MMU space ID of the specified process.

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

**Return Value Type**: Hex value.

## TASK.PROC.THREADS() List of threads

Syntax: **TASK.PROC.THREADS(***<process_magic>***,***<thread_magic>***)**

Returns the next magic in the thread list of the specified process.

**Parameter and Description**:

| | |
|---|---|
| *<process_magic>* | **Parameter Type**: Decimal or hex or binary value. |
| *<thread_magic>* | **Parameter Type**: Decimal or hex or binary value.<br>Use zero as *<thread_magic>* for the first thread. |

**Return Value Type**: Hex value.

**Return Value and Description**:

| | |
|---|---|
| **-1** | Returns -1 if no further thread available. |
| *<thread_magic>* | Returns the next magic in list. |

## TASK.PROC.TTB() TTB of process

Syntax: **TASK.PROC.TTB(***<process_magic>***)**

Returns the translation table base address of the specified process.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.QVM.FORMAT() Machine ID of VM

| | |
|---|---|
| Syntax: | **TASK.QVM.FORMAT(*<qvm_magic>*)** |

Returns the (QNX internal) format of a VM as value.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.QVM.MAGIC() Magic number of VM

| | |
|---|---|
| Syntax: | **TASK.QVM.MAGIC(*<qvm_name>*)** |

Returns the magic number of the specified VM.

**Parameter Type**: String.

**Return Value Type**: Hex value.

## TASK.QVM.MID() Machine ID of VM

| | |
|---|---|
| Syntax: | **TASK.QVM.MID(*<qvm_magic>*)** |

Returns the machine ID of the specified VM.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.QVM.NAME() Name of VM

| | |
|---|---|
| Syntax: | **TASK.QVM.NAME(*<qvm_magic>*)** |

Returns the name of the specified VM.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: String.

| Syntax: | **TASK.QVM.VMLIST(***<qvm_magic>***)** |
|---|---|

Returns the first or next magic in the VM list.

**Parameter and Description**:

| | |
|---|---|
| *<qvm_magic>* | **Parameter Type**: Decimal or hex or binary value. Use zero as *<qvm_magic>* to get the magic of the first VM. |

**Return Value and Description**:

| | |
|---|---|
| *<qvm_magic>* | Returns the next magic in the VM list. |
| **0** | Returns 0 if no further VM available. |

# Appendix

## Appendix A: Kernel debug information

In QNX version 6.5 and 6.6, the standard installation does not include debug information of the kernel, i.e. you will not be able to see the internal structures of a process or thread. The QNX awareness does not need this, so it's sufficient to use the standard kernel. However, if you want access to these internal structures, you have to install and use the debug version. Please follow this sequence to create kernel symbol files:

1. Locate the debug info files in the QNX SDP installation media, in the subdirectory “debugging_info”, or download "QNX Software Development Platform 6.x.x [Build xxxxxxxxxxxx] - Full Installation Debug Info Tar [For Reduced DVD]" from the QNX developer network download site.
2. Extract your target architecture's (e.g. “armle”) debug files to a temporary directory.
3. Copy the *<arch>*/boot/sys/procnto*-xxxxxxxxxxxx.sym files to the QNX installation directory target/qnx6/*<arch>*/boot/sys/ and remove the .sym extension from these files.
4. Open your system builder project (project.bld) and set the “System” properties “Create startup sym file?” to “Yes”, “Create proc sym file?” to “Yes” and for “Procnto” select the “procnto*-xxxxxxxxxxxx” file.
5. Rebuild the image.