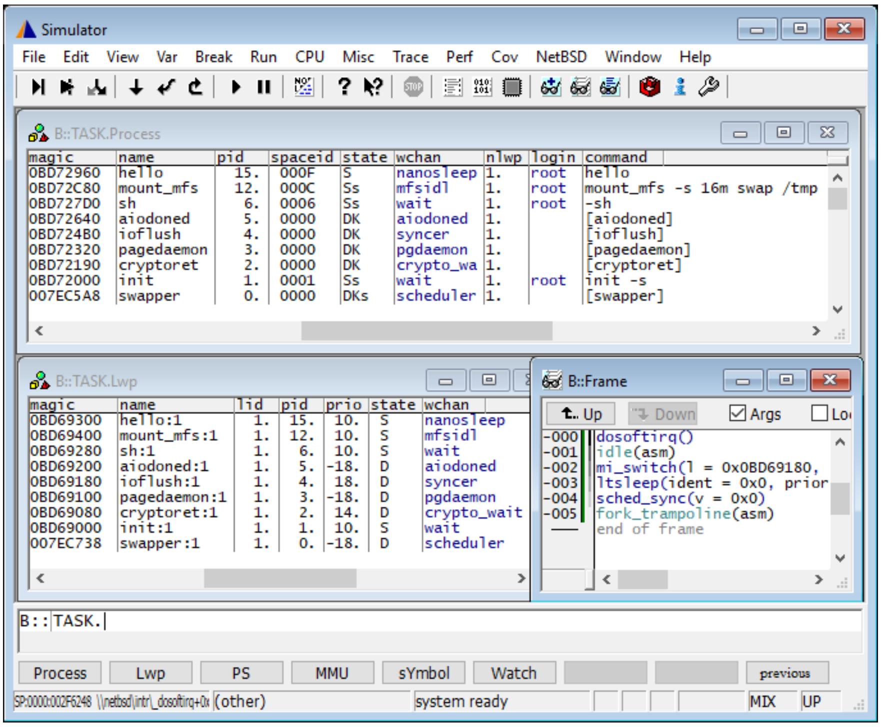
OS Awareness Manual NetBSD

Version 06-Jun-2024

Overview


Simulator
File Edit View Var Break Run CPU Misc Trace Perf Cov NetBSD Window Help
Process
Lwp
PS
MMU
sYmbol
Watch
previous

```
B::TASK.Process
magic    name        pid  spaceid state wchan      nlwp login command
0BD72960 hello       15.  000F    S     nanosleep  1.   root  hello
0BD72C80 mount_mfs   12.  000C    Ss    mfsidl     1.   root  mount_mfs -s 16m swap /tmp
0BD727D0 sh           6.  0006    Ss    wait       1.   root  -sh
0BD72640 aiodoned     5.  0000    DK    aiodoned   1.         [aiodoned]
0BD724B0 ioflush      4.  0000    DK    syncer     1.         [ioflush]
0BD72320 pagedaemon   3.  0000    DK    pgdaemon   1.         [pagedaemon]
0BD72190 cryptoret    2.  0000    DK    crypto_wa  1.         [cryptoret]
0BD72000 init         1.  0001    Ss    wait       1.   root  init -s
007EC5A8 swapper      0.  0000    DKs   scheduler  1.         [swapper]
```

```
B::TASK.Lwp
magic    name          lid  pid  prio state wchan
0BD69300 hello:1        1.  15.  10.  S     nanosleep
0BD69400 mount_mfs:1    1.  12.  10.  S     mfsidl
0BD69280 sh:1           1.   6.  10.  S     wait
0BD69200 aiodoned:1     1.   5. -18.  D     aiodoned
0BD69180 ioflush:1      1.   4.  18.  D     syncer
0BD69100 pagedaemon:1   1.   3. -18.  D     pgdaemon
0BD69080 cryptoret:1    1.   2.  14.  D     crypto_wait
0BD69000 init:1         1.   1.  10.  S     wait
007EC738 swapper:1      1.   0. -18.  D     scheduler
```

```
B::Frame
Up  Down  [✓ Args]  [ ] Lo
-000 dosoftirq()
-001 idle(asm)
-002 mi_switch(l = 0x0BD69180,
-003 ltsleep(ident = 0x0, prior
-004 sched_sync(v = 0x0)
-005 fork_trampoline(asm)
     end of frame
```

```
B::TASK.
```

```
SP:0000:002F6248 \\netbsd\intr\_dosoftirq+0x (other)   system ready   MIX   UP
```

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

Terminology

NetBSD uses the terms “processes” and “light weight processes” (LWPs). If not otherwise specified, the TRACE32 term “task” corresponds to NetBSD LWPs.

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 NetBSD is supported for the following versions:

- NetBSD 3.x and 4.x on ARM and PowerPC

Configuration

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

Automatic configuration tries to locate the NetBSD 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 netbsd**

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 example “~~/demo/*<processor>*/kernel/netbsd/netbsd.cmm”.

Quick Configuration Guide

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

1. Carefully read the PRACTICE demo start-up script (~~/demo/*<processor>*/kernel/netbsd/netbsd.cmm).

2. Make a copy of the PRACTICE script file “netbsd.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.

Now you can access the NetBSD extensions through the menu.

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

Hooks & Internals in NetBSD

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global symbols of the NetBSD kernel. That means, you have to compile the kernel with debug information:

In the kernel configuration file (usr/src/sys/arch/*<project>*/conf/*<board>*) , includes the line:

```
makeoptions     DEBUG="-g”      # compile full symbol table
```

In usr/src/distrib/*<project>*/ramdisk/Makefile and in usr/src/distrib/*<project>*/ramdisk/ramdiskbin.mk change

```
DBG= -Os
```

to

```
DBG= -g
```

The compiled kernel with debug info should then be available in:

usr/src/sys/arch/*<project>*/compile/*<board>*/netbsd.gdb.

Ensuer, that every time, when features of the OS Awareness are used, the symbols of “netbsd.gdb” are available and accessible.

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

Features

The OS Awareness for NetBSD supports the following features.

Display of Kernel Resources

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

TASK.LWP	LWPs
TASK.Process	Processes

For a description of the commands, refer to chapter "**NetBSD 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-Related Breakpoints

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

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

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

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

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

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

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

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

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

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

Task Context Display

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

To display a specific task context, use the command:

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

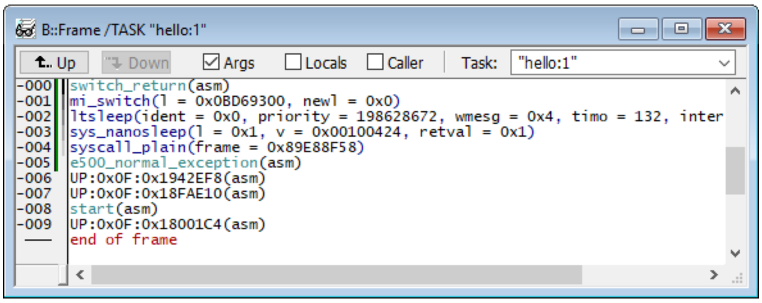
- Use a magic number, task ID, or task name for <*task*>. For information about the parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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

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

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

1. Open the **Frame /Caller /Task** <*task*> window.
2. Double-click the line showing the OS service call.



```
-000 switch_return(asm)
-001 mi_switch(l = 0x0BD69300, newl = 0x0)
-002 ltsleep(ident = 0x0, priority = 198628672, wmesg = 0x4, timo = 132, inter
-003 sys_nanosleep(l = 0x1, v = 0x00100424, retval = 0x1)
-004 syscall_plain(frame = 0x89E88F58)
-005 e500_normal_exception(asm)
-006 UP:0x0F:0x1942EF8(asm)
-007 UP:0x0F:0x18FAE10(asm)
-008 start(asm)
-009 UP:0x0F:0x18001C4(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 commands refer to this necessity.

Space IDs

Processes of NetBSD may reside virtually on the same address. To distinguish those addresses, the Debugger uses an additional space ID that specifies to which virtual memory space the address refers. The command **SYStem.Option.MMUSPACES ON** enables the additional space ID. For all processes using the kernel address space, the space ID is zero. For processes using their own address space, the space ID equals the process ID.

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.spaceid("*<process>*")** returns the space ID for a given process. If the space ID is not equal to zero, load the symbols of a process to this space ID:

```
LOCAL &spaceid
&spaceid=task.proc.spaceid("myProcess")
Data.LOAD myProcess &spaceid:0 /NoCODE /NoClear
```

See also chapter "**Debugging User Processes**".

Scanning System and Processes

The kernel code, which resides in the kernel space, can be accessed by any process, regardless of the current space ID. The command **TRANSlation.COMMON** defines those commonly used areas.

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

```
TASK.MMU.SCAN "swapper"
```

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

See also chapter "**Debugging NetBSD Kernel and User Processes**".

Dynamic Task Performance Measurement

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

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

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

Task Runtime Statistics

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

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

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

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

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

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)".

NetBSD specific Menu

The menu file "netbsd.men" contains a menu with NetBSD specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- The **Display** menu items launch the kernel resource display windows.
- The **Process Debugging** > **Symbols** menu items (if available) load and delete symbols of processes.
- The **Process Debugging** > **Watch Processes** submenu opens a window to watch for process starts and symbols.

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 thread runtime statistics, thread related function runtime statistics or statistics on thread states.

Debugging NetBSD Kernel and User Processes

NetBSD 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 NetBSD 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 (swapper) is attached to the space ID zero. Each process that has its own memory space gets a space ID that corresponds to its process ID.

See also chapter “**MMU Support**”.

NetBSD Kernel

The NetBSD system builder generates an ELF file, that contains the startup code and the kernel.

Additionally, the NetBSD Awareness needs the symbols of the NetBSD kernel. See “Hooks and Internals” how to generate debug information to the kernel.

Downloading the NetBSD Image

If you start the NetBSD image from Flash, or if you download the image via NFS, do this as you are doing it without debugging.

If you want to download the NetBSD image using the debugger, you have to watch about the file format. If the image is in ELF format, simply download this to the target. If the image 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 netbsd-myboard                 ; downloading ELF
Data.Load.Binary netbsd.uImage 0x00800000    ; downloading binary
```

When downloading the kernel via the debugger, remember to set startup parameter, that the kernel require, before booting the kernel. Usually the boot loader passes these parameters to the image.

Debugging the Kernel

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

```
Data.Load.Elf netbsd.gdb /NoCODE
```

You have to inform the debugger about the kernel address translations. Scan the processor MMU right after it is switched on:

```
MMU.SCAN
```

The kernel address space is visible to all processes, so specify the address range to be common to all space IDs. Check the example scripts for the right addresses:

```
TRANSlation.COMMON 0x0--0x00FFFFFF
```

And switch on the debugger MMU translation:

```
TRANSlation.ON
```

User Processes

Each user process in NetBSD gets its own virtual memory space. To distinguish the different memory spaces, the debugger assigns a “space ID”, which correlates 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 NetBSD 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.

Debugging the Process

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 and to scan the process’ MMU settings.

Manually Load Process Symbols:

For example, if you’ve got a a process called “hello” with the space ID 0xF:

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

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

Additionally, you have to scan the MMU translation table of this process:

```
TASK.MMU.SCAN 0xF         ; scan MMU of process ID 15.
```

It is possible, to scan the translation tables of all processes at once. On some processors, and depending on your number of active processes, this may take a very long time. In this case use the scanning of single processes, mentioned above. Scanning all processes:

```
TASK.MMU.SCAN    ; scan MMU entries of all processes
```

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. Set a breakpoint into the process start handler of NetBSD, when the process is already loaded but not yet startet. The function doexechooks() may serve as a good point. When the breakpoint is hit, check if the process is already loaded. If so, extract the space ID, load the symbols and scan the process MMU. Set a breakpoint to the main() routine of the process. As soon as the process is startet, the breakpoint will be hit. The following script shows an example of how to do this:

```
if run()
   Break

; Use conditional breakpoint to halt only, if “hello” is started
 Break.Set doexechooks /CONDition task.proc.space("hello")!=0xffffffff
 Go
 wait !run()          ; wait for the breakpoint to be hit (process start)
 Break.Delete doexechooks                    ; remove “helper” breakpoint


local &spaceid       ; PRACTICE macro holding the space ID of the process
 &spaceid=task.proc.spaceid("hello")                         ; get the space ID

 TASK.MMU.SCAN &spaceid                        ; scan MMU pages of new process

 Data.LOAD.Elf hello &spaceid:0 /NoCODE /NoClear           ; load symbols

 Break.Set main /Onchip              ; set breakpoint on main entry point

 Go                                          ; let Linux start the process
 wait !run()                                         ; will halt at main()
 Break.Delete main


; Now scan the MMU for new (swapped in) pages
 TASK.MMU.SCAN &spaceid
```

When finished debugging with a process, or if restarting the process, you have to delete the symbols and restart the application debugging. Delete the symbols with this command:

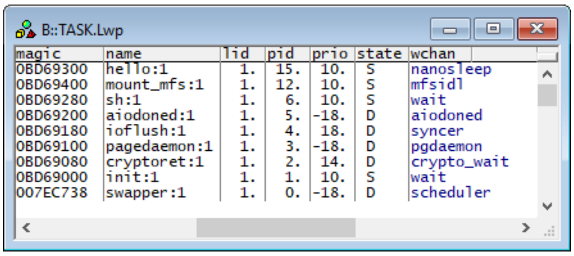
```
sYmbol.Delete \\hello
```

TASK.LWP Display LWPs

Format: **TASK.LWP** [<*lwp*>]

Displays the LWP table of NetBSD or detailed information about one specific LWP.

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



```
magic     name          lid  pid  prio  state  wchan
0BD69300  hello:1        1.  15.   10.  S      nanosleep
0BD69400  mount_mfs:1    1.  12.   10.  S      mfsidl
0BD69280  sh:1           1.   6.   10.  S      wait
0BD69200  aiodoned:1     1.   5.  -18.  D      aiodoned
0BD69180  ioflush:1      1.   4.   18.  D      syncer
0BD69100  pagedaemon:1   1.   3.  -18.  D      pgdaemon
0BD69080  cryptoret:1    1.   2.   14.  D      crypto_wait
0BD69000  init:1         1.   1.   10.  S      wait
007EC738  swapper:1      1.   0.  -18.  D      scheduler
```

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

The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

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 If no argument is specified, the command scans all current processes.

Example:

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

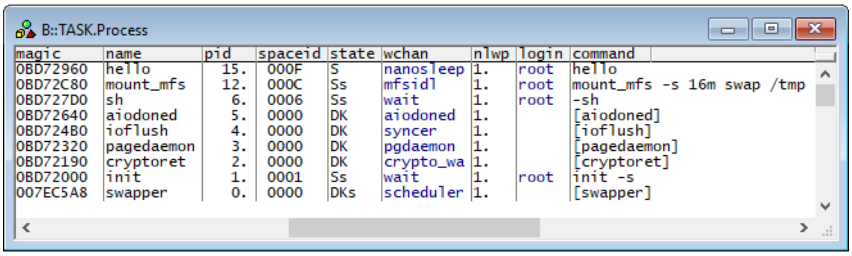
See also **MMU Support**.

TASK.Process Display processes

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

Displays the process table of NetBSD 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     name       pid  spaceid state wchan      nlwp login command
0BD72960  hello      15.   000F   S     nanosleep  1.   root  hello
0BD72C80  mount_mfs  12.   000C   Ss    mfsidl     1.   root  mount_mfs -s 16m swap /tmp
0BD727D0  sh          6.   0006   Ss    wait       1.   root  -sh
0BD72640  aiodoned    5.   0000   DK    aiodoned   1.         [aiodoned]
0BD724B0  ioflush     4.   0000   DK    syncer     1.         [ioflush]
0BD72320  pagedaemon  3.   0000   DK    pgdaemon   1.         [pagedaemon]
0BD72190  cryptoret   2.   0000   DK    crypto_wa  1.         [cryptoret]
0BD72000  init        1.   0001   Ss    wait       1.   root  init -s
007EC5A8  swapper     0.   0000   DKs   scheduler  1.         [swapper]
```

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

The field "magic" is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

NetBSD PRACTICE Functions

There are special definitions for NetBSD specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

Syntax: **TASK.CONFIG(magic | magicsize)**

Parameter and Description:

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

Return Value Type: Hex value.

TASK.PROC.SPACEID() Space ID of process

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

Returns the MMU space ID of the specified process.

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

Return Value Type: Hex value.