LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual LynxOS

OS Awareness Manual LynxOS

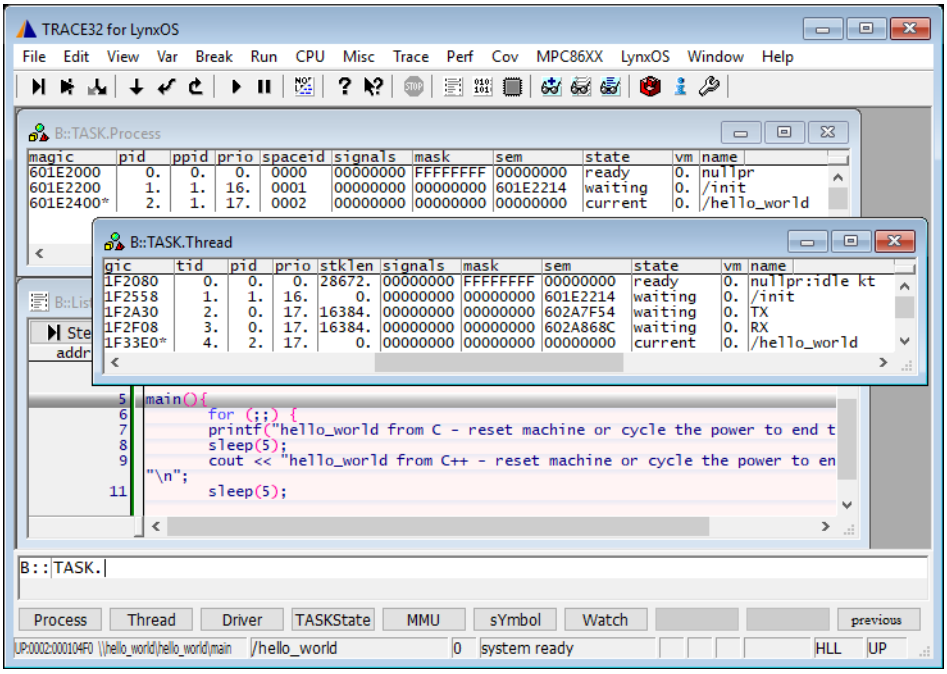
OS Awareness Manual LynxOS

Version 06-Jun-2024

History

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

Overview


TRACE32 for LynxOS
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC86XX LynxOS Window Help
B::TASK.Process
B::TASK.Thread
B::List
Step
addr
Process Thread Driver TASKState MMU sYmbol Watch previous
UP:0002:000104F0 \\hello_world\hello_world\main
/hello_world
0
system ready
HLL
UP

```
magic      pid  ppid prio spaceid signals  mask     sem      state    vm name
601E2000    0.   0.   0.  0000    00000000 FFFFFFFF 00000000 ready    0. nullpr
601E2200    1.   1.  16.  0001    00000000 00000000 601E2214 waiting  0. /init
601E2400*   2.   1.  17.  0002    00000000 00000000 00000000 current  0. /hello_world
```

```
gic      tid  pid  prio stklen signals  mask     sem      state    vm name
1F2080    0.   0.   0.  28672. 00000000 FFFFFFFF 00000000 ready    0. nullpr:idle kt
1F2558    1.   1.  16.     0.  00000000 00000000 601E2214 waiting  0. /init
1F2A30    2.   0.  17.  16384. 00000000 00000000 602A7F54 waiting  0. TX
1F2F08    3.   0.  17.  16384. 00000000 00000000 602A868C waiting  0. RX
1F33E0*   4.   2.  17.     0.  00000000 00000000 00000000 current  0. /hello_world
```

```
main(){
        for (;;) {
        printf("hello_world from C - reset machine or cycle the power to end t
        sleep(5);
        cout << "hello_world from C++ - reset machine or cycle the power to en
"\n";
        sleep(5);
```

```
B::TASK.
```

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

Terminology

LynxOS uses the terms “processes” and “threads”. If not otherwise specified, the TRACE32 term “task” corresponds to LynxOS 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 LynxOS is supported for the following versions:

- LynxOS Release 3.1, 4.0 and 5.0 on PowerPC.

Configuration

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

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

If you want to 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. 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 other arguments:

Format:	**TASK.CONFIG lynx**

See also the example “~~/demo/<*processor*>/kernel/lynx/lynx.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/lynx/lynx.cmm).

2. Make a copy of the PRACTICE script file “lynx.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).

Hooks & Internals in LynxOS

No hooks are used in the kernel.

For detecting the current running thread, the kernel symbol “currtptr” is used.

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols. Ensure that access to those symbols is possible every time when features of the OS Awareness are used. See chapter “**Debugging The Kernel**” how to load the kernel symbols into the debugger.

Features

The OS Awareness for LynxOS supports the following features.

Display of Kernel Resources

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

TASK.Process	Processes
TASK.Thread	Threads
TASK.Driver	Drivers

For a description of the commands, refer to chapter “**LynxOS 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.

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.

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 LynxOS 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 equaled 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.space("*<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.space("myProcess")
Data.LOAD myProcess &spaceid:0 /NoCODE /NoClear
```

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

Scanning System and Processes

PowerPC 860 type MMU:
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 (LynxOS calls them "job id"). Note that on some systems, this may take a long time. In this case you may scan single processes (see below).

PowerPC 603e type MMU:
The 603e-type MMU has an address translation that cannot be scanned fully automatically. However, the current used memory areas can be scanned with **MMU.SCAN BAT** and **MMU.SCAN PTE** (LynxOS uses both, BATs and PTEs).

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 0.
```

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

See also chapter "**Debugging LynxOS Kernel and User Processes**"

Symbol Autoloader

The OS Awareness for LynxOS contains an autoloader, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding LynxOS components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader, the debugger invokes the appropriate command. The command is usually a call to a PRACTICE script that loads the symbol file to the appropriate addresses.

The command **sYmbol.AutoLOAD.List** shows a list of all known address ranges/components and their symbol load commands.

The autoloader can be configured to react only on processes, drivers, (all) libraries, or libraries of the current process (see also **TASK.sYmbol.Option AutoLoad**). It is recommended to set only those components you are interested in, because this decreases the time of the autoloader checks highly.

The autoloader reads the target's tables for the chosen components and fills the autoloader list with the components found on the target. All necessary information, such as load addresses and space IDs, are retrieved from kernel-internal information.

sYmbol.AutoLOAD.CHECKLYNXOS "*<action>*"

<action>	Action to take for symbol load, e.g. `"DO autoload"`

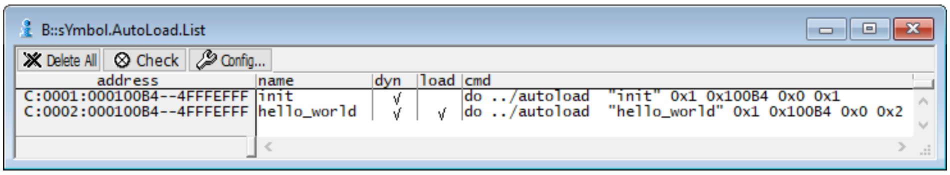
If an address is accessed that is covered by the autoloader list, the autoloader calls *<action>* and appends the load addresses and the space ID of the component to the action. Usually, *<action>* is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example script "autoload.cmm" in the ~~/demo directory.

The point in time when the component information is retrieved from the target can be set:

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

(no argument)	A single **sYmbol.AutoLOAD.CHECK** command refreshes the information about the target.
ON	The debugger automatically reads the information on every go/halt or step cycle. This significantly slows down the debugger's speed when single stepping.
ONGO	The debugger automatically reads the information on every go/halt cycle, but not when single stepping.
OFF	no automatic update of the autoloader table will be done, you have to manually trigger the information read when necessary. To accomplish that, execute the **sYmbol.AutoLOAD.CHECK** command without arguments.

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



```
         address          |name       |dyn |load |cmd
 C:0001:000100B4--4FFFEFFF |init       | √  |     |do ../autoload  "init" 0x1 0x100B4 0x0 0x1
 C:0002:000100B4--4FFFEFFF |hello_world| √  |  √  |do ../autoload  "hello_world" 0x1 0x100B4 0x0 0x2
```

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)".

LynxOS specific Menu

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

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

- The **Display** menu items launch the kernel resource display windows.
- The **Process Debugging** > **Symbols** menu items 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 LynxOS Kernel and User Processes

LynxOS runs on virtual address spaces. The kernel uses a static address translation, usually starting from virtual address 0xA0000000 mapped to the physical start address of the RAM. Each user process gets its own user address space when loaded, usually starting from virtual 0x0, mapped to any physical RAM area, that is currently free. Due to this address translations, debugging the LynxOS 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 is attached to the space ID zero. Each process that has its own memory space gets a space ID that is equal to its process ID.

See also chapter “**MMU Support**”.

LynxOS Kernel

When building a LynxOS KDI, the LynxOS make process links the kernel and BSP to a file called “a.out” and then generates the KDI. Preserve both files for using the LynxOS awareness.

Downloading The Kernel

If you start the LynxOS kernel from Flash, or if you download the kernel via Ethernet, do this as you are doing it without debugging.

If you want to download the kernel image using the debugger, you have to specify, to which address to download it. The LynxOS kernel image is usually located at the physical start address of the RAM, after the vector table.

When downloading a KDI image, specify the start address, where to load. E.g., if the physical address starts at 0x0, download it to 0x4000, skipping the vector table:

```
Data.LOAD.Binary hello.kdi 0x4000 /NosYmbol
```

After downloading with the debugger, set the program counter to the physical start address. If the download address is 0x4000, the start address is usually 0x4020.

When downloading the kernel via the debugger, remember to set startup options, that the kernel may require, before booting the kernel.

Debugging The Kernel Startup

The kernel image starts with MMU switched off, i.e. the processor operates on physical addresses. However, all symbols of the kernel are virtual addresses. If you want to debug this (tiny) startup sequence, you have to load and relocate the symbols.

- Downloading the kernel via debugger:

  Download the KDI as mentioned above, then load and relocate the symbols as mentioned below.

- Downloading the kernel via boot loader:

  Just load the symbols into the debugger *before* it is downloaded by the boot loader as mentioned below.

  Then, set an on-chip(!) breakpoint to the physical start address of the kernel (software breakpoints won't work, as they would be overwritten by the kernel download):

  ```
  Break.Set 0x4020 /Onchip
  ```

  Now let the boot loader download and start the LynxOS KDI. It will halt on the start address, ready to debug. Delete the breakpoint when hit.

While the MMU is switched off, you have to load and relocate the symbols to the physical addresses. E.g., if physical address is 0x00000000 and virtual address is 0xA0000000:

```
Data.LOAD.XCOFF a.out /NoCODE           ; load the symbols

sYmbol.RELOC C:0x00000000-0xa0000000    ; relocate them
```

As soon as the processor MMU is switched on, you have to reload the symbol to it's virtual addresses. 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 LynxOS awareness, you have to load the virtual addressed symbols of the kernel into the debugger. The kernel symbols reside in the kernel image, which is later linked into a KDI. The kernel image is usually called “a.out” and can be found in the BSP directory. The a.out XCOFF image contains all addresses in virtual format, so it’s enough to simply load the file:

```
Data.LOAD.XCOFF a.out /NoCODE
```

Next, scan the processor MMU right after it is switched on:

```
MMU.SCAN                                ; or MMU.BATSCAN/PTESCAN if 603
                                        ; type MMU
```

The kernel address space is visible to all processes, so specify the address range to be common to all space IDs:

```
TRANSlation.COMMON 0xA0000000--0xFFFFFFFF
```

And switch on the debugger MMU translation:

```
TRANSlation.ON
```

User Processes

Each user process in LynxOS gets its own virtual memory space, each usually starting at address zero. To distinguish the different memory spaces, the debugger assigns a “space ID”, which is 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 LynxOS 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.

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 process ID **`12.`** (the dot specifies a decimal number!):

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

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

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

```
TASK.MMU.SCAN 12.                     ; scan MMU of process ID 12.
```

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
; or:
MMU.SCAN ALL                          ; this one's faster
```

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 and scan MMU
```

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

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 LynxOS, when the process is already loaded but not yet started. The functions .execuser() (if available) may serve as a good point. When the breakpoint is hit, check if the process is already loaded. If so, extract the process ID, and load the symbols. Scan the process’ MMU and set a breakpoint to the main() routine of the process. As soon as the process is started, the breakpoint will be hit. The following script shows an example of how to do this:

```
LOCAL &spaceid          ; variable holding the space ID of the process

IF RUN()
    Break

Break.Set `.execuser` /CONDition task.proc.space("hello")!=0xffffffff
Go
WAIT !RUN()
Break.Delete `.execuser`

&spaceid=task.proc.space("hello")
TASK.MMU.SCAN &spaceid
Data.LOAD.Elf hello &spaceid:0 /cpp /NoCODE /NoClear

Break.Set main

Go                      ; let LynxOS start the process
WAIT !RUN()             ; will halt at main()
Break.Delete main
```

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.

TASK.Driver

Display drivers

Format: **TASK.Driver** [*<driver>*]

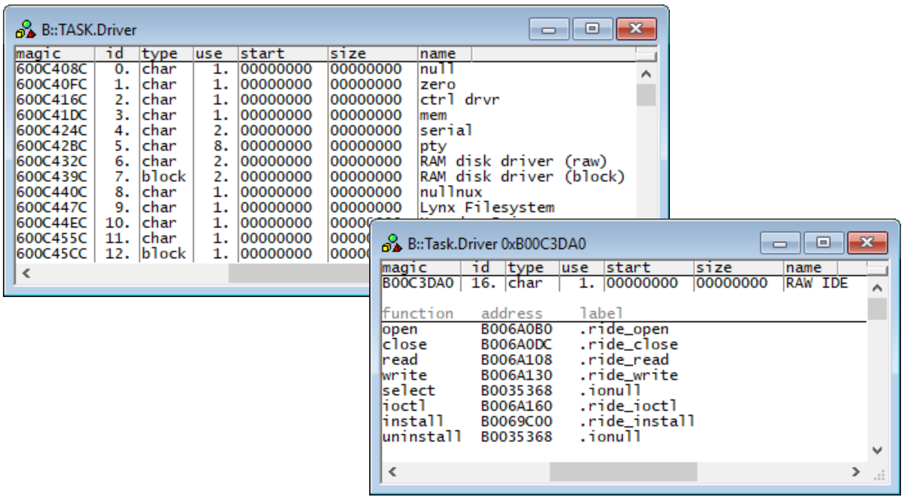
Displays the driver table of LynxOS or detailed information about one specific driver.

The display is similar to the LynxOS call “drivers”.

Without any arguments, a table with all created drivers will be shown.
Specify a driver name, ID or magic number to display information on only that driver.

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

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



```
magic     id   type  use   start     size      name
600C408C   0.  char    1.  00000000  00000000  null
600C40FC   1.  char    1.  00000000  00000000  zero
600C416C   2.  char    1.  00000000  00000000  ctrl drvr
600C41DC   3.  char    1.  00000000  00000000  mem
600C424C   4.  char    2.  00000000  00000000  serial
600C42BC   5.  char    8.  00000000  00000000  pty
600C432C   6.  char    2.  00000000  00000000  RAM disk driver (raw)
600C439C   7.  block   2.  00000000  00000000  RAM disk driver (block)
600C440C   8.  char    1.  00000000  00000000  nullnux
600C447C   9.  char    1.  00000000  00000000  Lynx Filesystem
600C44EC  10.  char    1.  00000000  0000
600C455C  11.  char    1.  00000000  0000
600C45CC  12.  block   1.  00000000  0000
```

```
magic     id   type  use   start     size      name
B00C3DA0  16.  char    1.  00000000  00000000  RAW IDE

function   address   label
open       B006A0B0  .ride_open
close      B006A0DC  .ride_close
read       B006A108  .ride_read
write      B006A130  .ride_write
select     B0035368  .ionull
ioctl      B006A160  .ride_ioctl
install    B0069C00  .ride_install
uninstall  B0035368  .ionull
```

TASK.MMU.SCAN Scan process MMU space

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

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>	If a process magic, ID or name is specified If no argument is specified, the command scans all current processes.
<start_address> *<size>*	The optional start address and size parameter limit the scanning of the MMU tables to the specified area to increase performance.

Example:

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

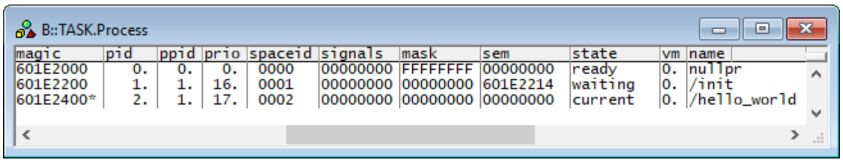
See also **MMU Support**.

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

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

The display is similar to the SKDB kernel debugger’s “p” dump of processes.

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



```
magic      pid   ppid prio spaceid signals  mask     sem      state    vm name
601E2000    0.    0.   0.  0000   00000000 FFFFFFFF 00000000 ready    0. nullpr
601E2200    1.    1.  16.  0001   00000000 00000000 601E2214 waiting  0. /init
601E2400*   2.    1.  17.  0002   00000000 00000000 00000000 current  0. /hello_world
```

“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.

TASK.sYmbol   Process symbol management

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

TASK.sYmbol.LOAD	Load process symbols and MMU
TASK.sYmbol.DELete	Unload process symbols and MMU
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 and deletes its MMU entries.

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

Example: When deleting the above loaded symbols with the command:

```
TASK.sYmbol.DELete "hello"
```

the debugger will internally execute the commands:

```
TRANSlation.Delete 6.:0--0xffffffff
sYmbol.Delete \\hello
```

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, and the process specific MMU settings (see chapter “Debugging User Processes”).
This command retrieves the appropriate space ID, loads the symbol file of an existing process and reads its MMU entries. Note that this command works only with processes that are already loaded in LynxOS (i.e. processes that show up in the **TASK.Process** window).

Example:
If the **TASK.Process** window shows a “/hello” process with process ID 6, the command:

```
TASK.sYmbol.LOAD "hello"
```

will internally execute the commands:

```
TASK.MMU.SCAN 6.
Data.LOAD.Elf hello 6.:0 /cpp /NoCODE /NoClear
```

The actual load command can be adjusted with **TASK.sYmbol.Option LOADCMD**

Format:	**TASK.sYmbol.Option** *<option>*
<option>:	**LOADCMD** *<command>*
	MMUSCAN ON | **OFF**
	AutoLoad *<option>*

Set a specific option to the symbol management.

LOADCMD:

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"
```

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 AutoLoader:

Process	check processes
Library	check all libraries of all processes
CurrLib	check only libraries of current process
DRiVer	check dynamically loaded drivers
ALL	check processes, and all libraries
NoProcess	don't check processes

NoLibrary	don’t check libraries
NoDRiVer	don’t check drivers
NONE	check nothing.

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

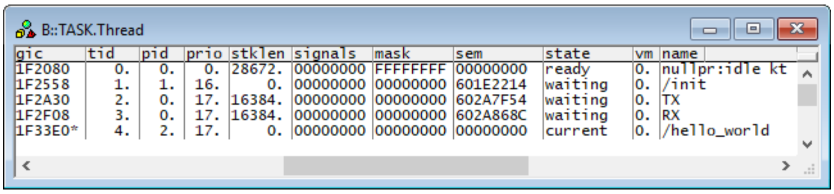
TASK.Thread

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

Displays the thread table of LynxOS or information about one specific thread.

The display is similar to the SKDB kernel debugger's "p" dump of threads.

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



```
B::TASK.Thread
gic     |tid |pid |prio|stklen|signals |mask    |sem     |state   |vm|name
1F2080  |  0.|  0.|  0.|28672.|00000000|FFFFFFFF|00000000|ready   |0.|nullpr:idle kt
1F2558  |  1.|  1.| 16.|    0.|00000000|00000000|601E2214|waiting |0.|/init
1F2A30  |  2.|  0.| 17.|16384.|00000000|00000000|602A7F54|waiting |0.|TX
1F2F08  |  3.|  0.| 17.|16384.|00000000|00000000|602A868C|waiting |0.|RX
1F33E0* |  4.|  2.| 17.|    0.|00000000|00000000|00000000|current |0.|/hello_world
```

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

The **TASK.Watch** command group builds a watch system that watches your LynxOS 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 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 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 breakpoints for detection of process creation.

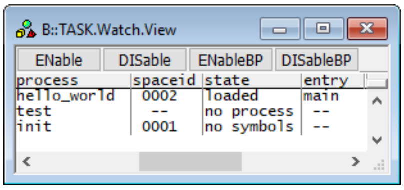
Please see **TASK.Watch.View** for details.

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.



```
process      spaceid state       entry
hello_world   0002   loaded      main
test           --    no process   --
init          0001   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 (= process ID) of the watched process. 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. If grayed, the debugger is currently not able to determine the watch state. **no process**: The debugger couldn't find the process in the current LynxOS process list. **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). **loaded**: The debugger found the process and loaded the process's MMU settings and symbols.
entry	The process entry point, which is `main()`. 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 and its MMU settings, 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	Add processes to the watch list.
TASK.Watch.DELete	Remove 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 scans its MMU entries and 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 LynxOS process list, the watch system unloads the symbols and removes the MMU settings from the debugger MMU table. 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 a 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 a 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. 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.

LynxOS PRACTICE Functions

There are special definitions for LynxOS 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.SPACE() Space ID of process

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

Returns the debugger MMU space ID of the specified process.

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

Return Value Type: Hex value.

TASK.DRIVER.START() Start address of driver

Syntax: **TASK.DRIVER.START("***<driver_name>***")**

Returns the start address of the specified driver.

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

Return Value Type: Hex value.

TASK.DRIVER.TEXT() Address of .text section

Syntax: **TASK.DRIVER.TEXT("***<driver_name>***")**

Returns the address of the .text section.

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

Return Value Type: Hex value.

TASK.DRIVER.DATA() Address of .data section

Syntax: **TASK.DRIVER.DATA("***<driver_name>***")**

Returns the address of the .data section.

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

Return Value Type: Hex value.

TASK.DRIVER.BSS() Address of .bss section

Syntax: **TASK.DRIVER.BSS("***<driver_name>***")**

Returns the address of the .bss section.

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

Return Value Type: Hex value.