LAUTERBACH
DEVELOPMENT TOOLS

Hypervisor Awareness Manual
Wind River Hypervisor

MANUAL

Hypervisor Awareness Manual Wind River Hypervisor

Hypervisor Awareness Manual Wind River Hypervisor

Version 06-Jun-2024

Overview

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

NOTE:	This documentation is **outdated** and will be replaced soon by a newer version. It is especially **not** suitable for Wind River Helix.

Terminology

The Wind River Hypervisor manages “virtual boards” and “threads”. If not otherwise specified, the TRACE32 term “task” corresponds to Hypervisor “thread”, while a Hypervisor “virtual board” corresponds to a “space ID” in TRACE32.

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 Wind River Hypervisor is supported for the following versions:

- Version 2.x on ARM, PowerPC and x64
- Version 3.x on ARM and x64

NOTE:	This document is **outdated** and will be replaced soon by a newer version. It is especially **not** suitable for Wind River Helix. For proper configuration please ask Lauterbach for assistance.

Configuration

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

Automatic configuration tries to locate the hypervisor internals automatically. For this purpose the symbols of the hypervisor kernel must be loaded and accessible at any time the Hypervisor 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 Hypervisor Awareness. For this purpose it is necessary that all system internal symbols are loaded and accessible at any time the Hypervisor 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/**<*arch*>**/kernel/wrhv/wrhv.t32**

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

Note that the symbols of the hypervisor must be loaded into the debugger. See **Hooks & Internals** for details on the used symbols. See also the examples in the demo directories “~~/demo/<*arch*>/kernel/wrhv”.

Quick Configuration Guide

To fully configure the Hypervisor Awareness for Wind River Hypervisor, please use one of the demo startup scripts as template. Find the templates in the directory ~~/demo/<*arch*>/kernel/wrhv.

Follow this roadmap:

1. Carefully read the demo startup scripts (~~/demo/<*arch*>/kernel/wrhv).
2. Make a copy of the appropriate script. Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the hypervisor resources and use the trace functions (if available).

Now you can access the Hypervisor extensions through the menu.

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

No hooks are used in the kernel.

For retrieving the kernel data structures, the Hypervisor Awareness uses the global kernel symbols of the Wind River Hypervisor. This means that every time, when features of the Hypervisor Awareness are used, the symbols of the hypervisor must be available and accessible.

The image project of your hypervisor application creates a symbol file called “hypervisor” in the object directory. Load the symbols to space ID zero with the command:

```
Data.LOAD.Elf <path_to_project>/obj/hypervisor 0:0 /NoCODE
```

“rootOS” guest image called “vxWorks”. The load procedure is a little bit complicated (see example scripts).

rootOS need to be loaded to machine ID one (1:::0).

Please also look at the demo start-up script wrhv.cmm, how to load the kernel symbols and the symbols of your application.

Features

The Hypervisor Awareness for Wind River Hypervisor supports the following features.

Display of Hypervisor Resources

The extension defines new commands to display various hypervisor resources. Information on the following Wind River Hypervisor components can be displayed:

TASK.ThrList	Hypervisor threads
TASK.VirtBoard	Virtual boards
TASK.SysInfo	System Information
TASK.ConfigVec	Configuration vector files
TASK.REGistry	Registry
TASK.CoreState	Hypervisor core state information

For a description of the commands, refer to chapter "**Wind River Hypervisor 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.

NOTE:	The stack coverage analysis will only show valid results for Hypervisor threads that are *not* bound to a virtual board.

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:

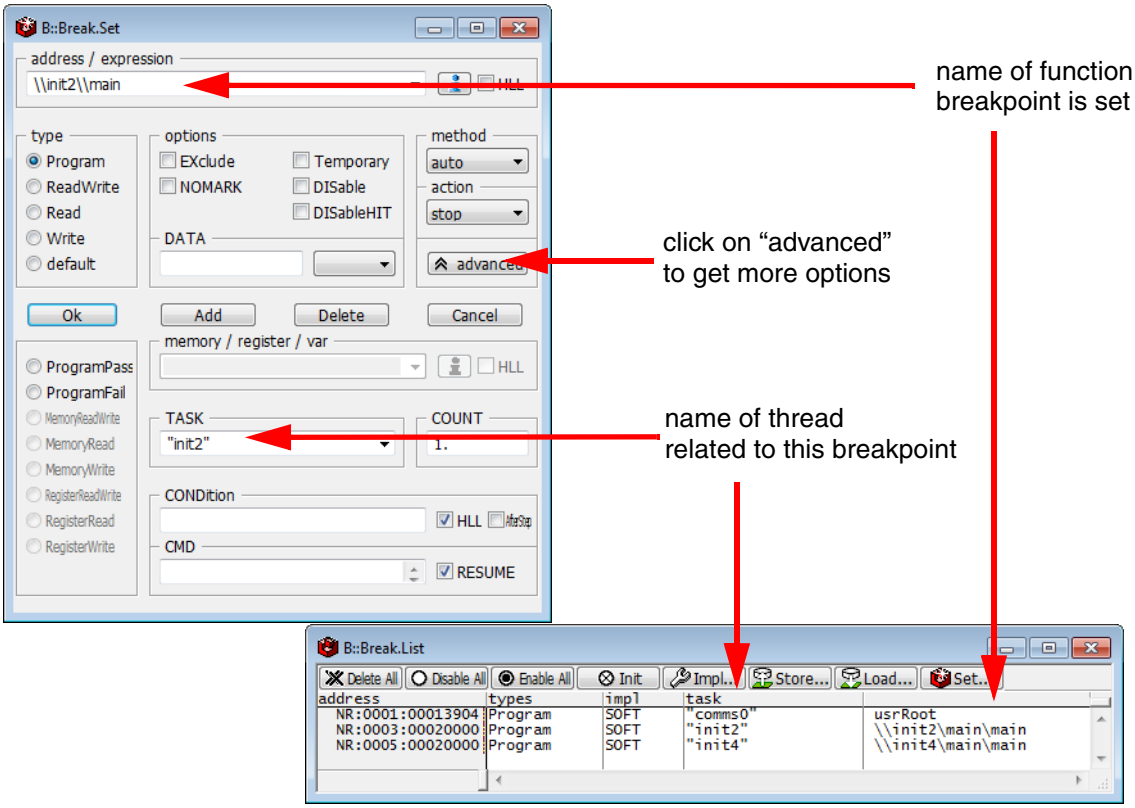
Break.CONFIG.UseContextID ON	Enables the comparison to the whole Context ID register.
Break.CONFIG.MatchASID ON	Enables the comparison to the ASID part only.
TASK.List.tasks	If **TASK.List.tasks** provides a trace ID (**traceid** column), the debugger will use this ID for comparison. Without the trace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



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.

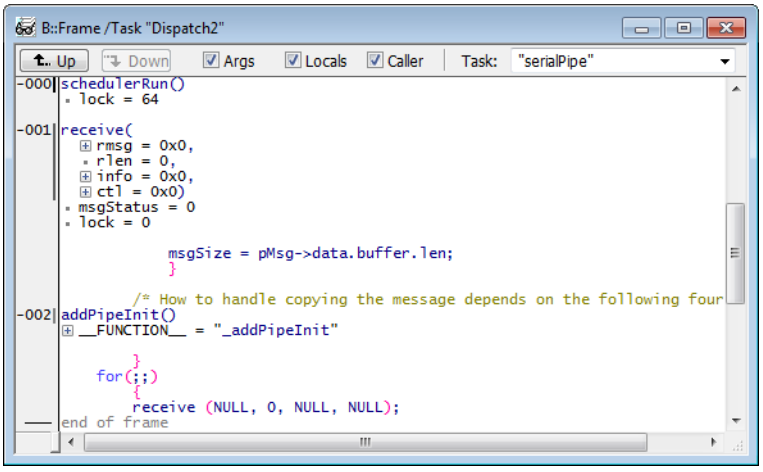
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.



Call stack frame of a thread, showing the calling line and local variables.

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.

ID, guest translations...).

Space IDs

Different virtual boards 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 hypervisor threads using the hypervisor address space, the space ID is zero. For virtual boards and their owning threads, the space ID corresponds to the board 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.virtboard.id(task.virtboard.magic("*<virtboard>*")** returns the ID for a given virtual board.

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
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
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 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.

<physical_kernel_address>

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

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

Example: ARM with 512 MB at physical address zero:

```
MMU.FORMAT STD hv_pageTable
TRANSlation.Create 0x0:0x0--0x1fffffff      ; hypervisor translation
TRANSlation.TableWalk ON
TRANSlation.ON
```

Example: x64 with EPT:

```
MMU.FORMAT EPT
TRANSlation.Create N:0x0:0x0--0xff7fffff    ; hypervisor translation
TRANSlation.TableWalk ON
TRANSlation.ON
```

Please see also the sample 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.

walk.

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 a single Virtual Board (see below).

To scan the address translation of a specific Virtual Board, use the command **MMU.SCAN TaskPageTable <space_id>:0.** . This command scans the space ID of the specified virtual board. E.g:

```
MMU.SCAN TaskPageTable 3:0
```

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 its 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 changes, 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 WR Hypervisor and Virtual Boards**".

Symbol Autoloader

The Hypervisor Awareness for Wind River Hypervisor contains a "Symbol Autoloader", which automatically loads symbol files corresponding to applications running in virtual boards. The autoloader maintains a list of address ranges, corresponding to virtual boards 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 hypervisor 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 virtual boards are started and stopped. The command **sYmbol.AutoLOAD.CHECK** configures the strategy, when to "check" the hypervisor data structures for changes in order to keep the debugger's information regarding the components up-to-date.

(no arguments)	The **sYmbol.AutoLOAD.CHECK** command *immediately* updates the component information by reading the hypervisor 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**).
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.
ONGO	The debugger checks for changed component info like with `ON`, but *not when performing single steps*.
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 task or library table are not covered.

When configuring the Hypervisor Awareness for Wind River Hypervisor, set up the symbol autoloader with the following command:

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

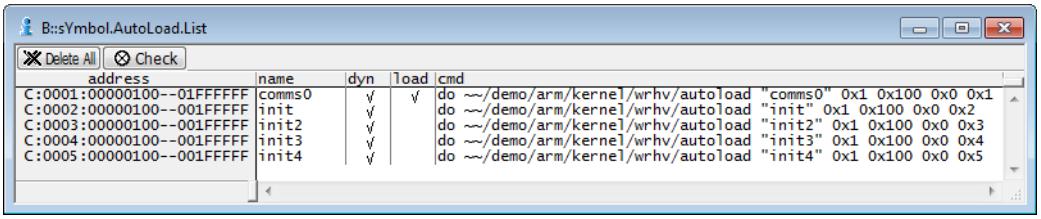
<action> action to take for symbol load, e.g. "`do autoload `"

The command **sYmbol.AutoLOAD.CHECKCoMmanD** 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 Wind River Hypervisor, the pre-defined action is to call the script ~~/demo/*<arch>*/kernel/wrhv/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 (virtual board), 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.



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.

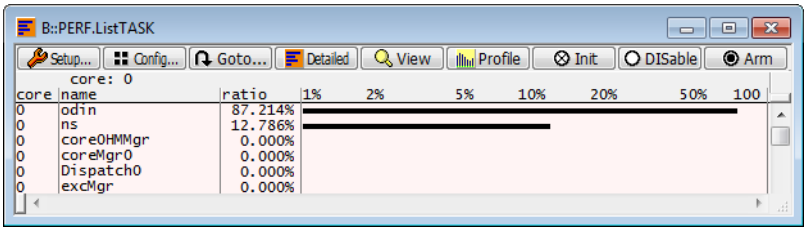
Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

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)".

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:

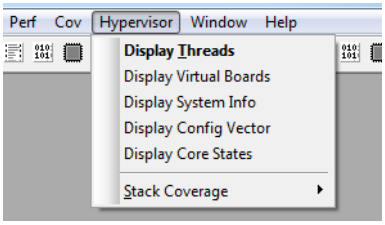
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat runtime timechart

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

Wind River Hypervisor specific Menu

The menu file “wrhv.men” contains a menu with Wind River Hypervisor specific menu items. Load this menu with the **MENU.ReProgram** command.

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



- The **Display** menu items launch the hypervisor resource display windows. See chapter “**Display of Hypervisor Resources**”.
- The **Stack Coverage** submenu starts and resets the Hypervisor specific stack coverage and provides an easy way to add or remove threads from the stack coverage window.
- Use the **Symbol 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.

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 or thread related function runtime statistics, if a trace is available. See also chapter “**Task Runtime Statistics**”.

Debugging Wind River Hypervisor Components

The hypervisor itself typically runs on physical addresses or with a static address translation. In contrast, each virtual board gets its own virtual address space when loaded, mapped to any physical RAM area that is currently free. Due to this address translations, debugging the hypervisor and the virtual boards requires some settings to the debugger.

To distinguish those different memory mappings, TRACE32 uses space IDs, defining individual address translations for each ID. The hypervisor is attached to the space ID zero. Each virtual board gets a space ID that corresponds to its board ID.

See also chapter "**MMU Support**".

Hypervisor

When building the image project, you typically get a "system.elf" file that contains the startup code, the hypervisor and any given virtual board with its applications.

Additionally, the Hypervisor Awareness needs the symbols of the hypervisor. Please see section "**Hooks & Internals**" how to find the symbol files of the hypervisor.

Downloading the image

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

If you want to download the hypervisor image using the debugger, simply download the generates "system.elf" file to the target. The target has to be initialized when downloading. Please also see the example scripts.

Example:

```
Data.Load.Elf system.elf                        ; downloading ELF image
```

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

Debugging the hypervisor

For debugging the hypervisor itself, and for using the Hypervisor Awareness, you have to load the symbols of the hypervisor into the debugger. The symbol file is usually named “hypervisor” and is placed in the object directory of the image project. Load the hypervisor symbols onto space ID zero.

E.g.:

```
Data.Load.Elf hypervisor 0:0 /NoCODE
```

Virtual Boards

Each virtual board in Wind River Hypervisor gets its own virtual memory space. To distinguish the different memory spaces, the debugger assigns a space ID, which correlates to the board ID. Using this space ID, it is possible to address a unique memory location, even if several virtual boards use the same virtual address.

Note that at every time the Hypervisor Awareness is used, it needs the hypervisor symbols. Please see the chapters above, how to load them. Hence, load all symbols of virtual boards with the option `/NoClear`, to preserve the hypervisor symbols.

Ensure that you load the symbol file containing debug information, i.e. the “unstripped” version.

Debugging a virtual board

To correlate the symbols of virtual board with the virtual addresses of this board, it is necessary to load the symbols into its space ID.

Manually Load Virtual Board Symbols:

For example, if you’ve got a a virtual board called “hello” with the board ID 12 (the dot specifies a decimal number!):

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

The board ID may also be calculated by using the PRACTICE functions **TASK.VIRTBOARD.MAGIC()** and **TASK.VIRTBOARD.ID()** (see chapter “**Wind River Hypervisor PRACTICE Functions**”).

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 virtual board. You can also force the loading of the symbols of a virtual board with

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

Using the Menus:

Select "Display Virtual Boards", right click on the "magic" of a virtual board, and select "Load symbols".

Start Debugging a virtual board from its entry point

The script "`wait_for_vb_start.cmm`" in the ~~/demo directory can be used to halt the debugger at the entry point of a virtual board, right after when it was created. Call the script with the name of the virtual board *first, then* start the board within the hypervisor. The script waits for the board to be started and halts the debugger at the entry point. You can then load the symbols of the virtual board as shown above.

Example:

```
; Wait for virtual board "hello" to be started

LOCAL &vb &spaceid
&vb="hello"

DO ~~/demo/arm/kernel/wrhv/wait_for_vb_start &vb

; Load the symbols of the virtual board to appropriate space ID

&spaceid=task.virtboard.id(task.virtboard.magic("&vb"))
Data.LOAD.ELF workspace/HIP/obj/&vb.elf &spaceid:0 /NoCODE /NoClear

; and "Go" to "main"

Go \\&vb\\main
```

TASK.ThrList — Display hypervisor threads

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

Displays the thread table of Wind River Hypervisor, or detailed information about one specific thread.

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

“magic” is a unique ID used by the Hypervisor Awareness to identify a specific thread (address of the context struct).

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

TASK.VirtBoard — Display virtual boards

Format: **TASK.VirtBoard** [<*board*>]

Displays the table of virtual boards or detailed information about one specific board.

Without any arguments, a table with all created virtual boards will be shown.
Specify a board magic number, ID or name to display detailed information on that board.

“magic” is a unique ID used by the Hypervisor Awareness to identify a specific virtual board (address of the board struct).

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

TASK.ConfigVec — Display configuration vector files

Format: **TASK.ConfigVec**

Displays a table with the configuration vector files defined in Wind River Hypervisor.
This command is only available on Hypervisor version 2.x.

TASK.REGistry — Display registry

Format: **TASK.REGistry**

Displays the registry tree of the system.
This command is only available on Hypervisor version 3.x.

TASK.SysInfo — Display system information

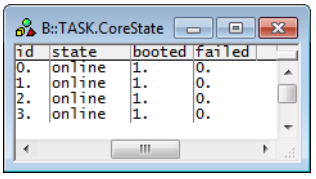
Format: **TASK.SysInfo**

Displays information about the hypervisor system.

TASK.CoreState — Display core information

Format: **TASK.CoreState**

Displays information about the hardware cores used by Wind River Hypervisor.



Wind River Hypervisor PRACTICE Functions

There are special definitions for Wind River Hypervisor specific PRACTICE functions.

TASK.CONFIG() 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 magic number in bytes.

Return Value Type: Hex value.

TASK.PRIV2HYP() Linear address

Syntax: **TASK.PRIV2HYP(***<address>***,***<cpu>***)**

Returns the linear address of the given CPU private address.

Parameter and Description:

<address>	**Parameter Type**: Decimal or hex or binary value.
<cpu>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.THREAD.ID() ID of thread

Syntax: **TASK.THREAD.ID(***<thread_magic>***)**

Returns the ID of the given thread.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.THREAD.MAGIC() Magic of thread

Syntax: **TASK.THREAD.MAGIC("***<thread_name>***")**

Returns the “magic” of the given thread.

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

Return Value Type: Hex value.

TASK.THREAD.PC() PC of thread

Syntax: **TASK.THREAD.PC(***<thread_magic>***)**

Returns the PC of the given thread.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.THREAD.TTB() TTB address of thread

Syntax: **TASK.THREAD.TTB(***<thread_magic>***)**

Returns the TTB address of the given thread.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.VIRTBOARD.BASE() Physical base address of virtual board

Syntax: **TASK.VIRTBOARD.BASE(***<board_magic>***)**

Returns the physical base address of the given virtual board.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.VIRTBOARD.ID() ID of virtual board

Syntax: **TASK.VIRTBOARD.ID(**<*board_magic*>**)**

Returns the ID of the given virtual board.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.VIRTBOARD.MAGIC() Magic of virtual board

Syntax: **TASK.VIRTBOARD.MAGIC("**<*board_name*>**")**

Returns the “magic” of the given virtual board.

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

Return Value Type: Hex value.

TASK.VIRTBOARD.START() Start address of virtual board

Syntax: **TASK.VIRTBOARD.START(**<*board_magic*>**)**

Returns the start address of the given virtual board.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.