LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
Nucleus PLUS

Release 09.2023

MANUAL

OS Awareness Manual Nucleus PLUS

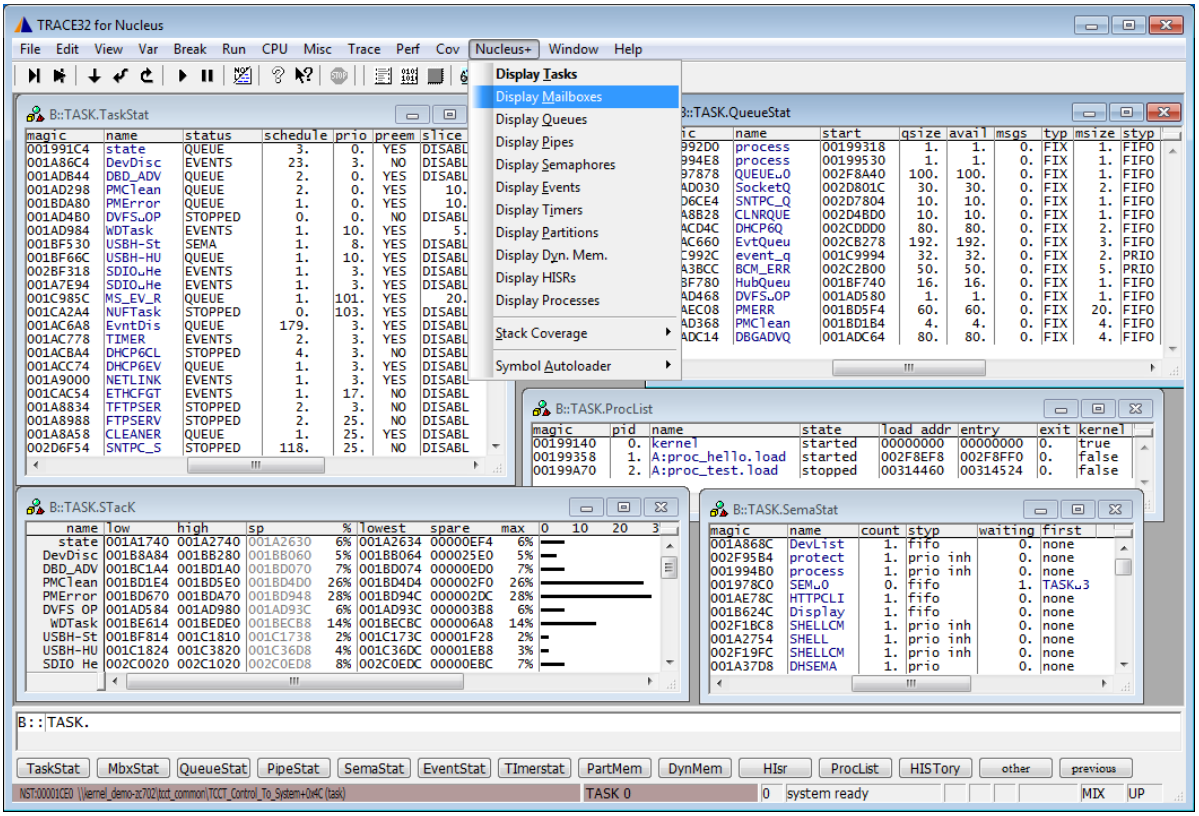
OS Awareness Manual Nucleus PLUS

Version 09-Oct-2023

History

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

Overview



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

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Training Basic Debugging"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

Supported Versions

Currently Nucleus is supported and tested for the following versions:

- Nucleus (PLUS) for ARM9/11, Cortex-M/R/A, ColdFire, Microblaze, MIPS, PowerPC;
- Nucleus Versions 1.x to 3.x, 2013.x, 2015.x

Configuration

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

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

If a system symbol is not available or if another address should be used for a specific system variable then the corresponding argument must be set manually with the appropriate address. In this case, use the manual configuration, which can require some additional arguments.

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).

Manual Configuration

Manual configuration for the OS Awareness for Nucleus can be used to explicitly define some memory locations. It is recommended to use automatic configuration.

Format:	**TASK.CONFIG nucleus** *<magic_address> <ticktime> <args>*

<magic_address>	Specifies a memory location that contains the current running task. This address can be found at the label “`TCD_Current_Thread`”.
<ticktime>	*<ticktime>* tells the OS Awareness, how many milliseconds one Nucleus PLUS tick has. This is currently only used in the **TASK.HISTory** command. (Don’t forget the dot for a decimal number!)

<args>	The additional arguments specify the symbols of the object lists. Use them as shown below:
	`TCD_Created_Tasks_List MDB_Created_Mailboxes_List`
	`QUD_Created_Queues_List PID_Created_Pipes_List`
	`SMD_Created_Semaphores_List`
	`EVD_Created_Event_Groups_List TMD_Created_Timers_List`
	`PMD_Created_Pools_List DMD_Created_Pools_List`

See also the example “~~/demo/*<arch>*/kernel/nucleus/nucleus.cmm”, which is a sample start-up script.

Automatic Configuration

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 nucleus**

If a system symbol is not available, or if another address should be used for a specific system variable, then the corresponding argument must be set manually with the appropriate address (see **Manual Configuration**).

See **Hooks & Internals** for details on the used symbols.

See also the example "~~/demo/<*arch*>/kernel/nucleus/nucleus.cmm", which is a sample start-up script.

Quick Configuration Guide

To get a quick access to the features of the OS Awareness for Nucleus with your application, follow this roadmap:

1. Start the TRACE32 Debugger.
2. Load your application as normal.
3. Execute the command:

   ```
   TASK.CONFIG ~~/demo/<arch>/kernel/nucleus/nucleus.t32
   ```

   See "**Automatic Configuration**".
4. Execute the command:

   ```
   MENU.ReProgram ~~/demo/<arch>/kernel/nucleus/nucleus.men
   ```

   See "**Nucleus Specific Menu**".
5. Start your application.

Now you can access the Nucleus extensions through the menu.

If you use Nucleus Processes, please also set up the **Symbol Autoloader**.

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 OS Awareness uses the global kernel symbols and structure definitions. Ensure that the kernel is compiled with debug information and that access to the kernel structures is possible every time when features of the OS Awareness are used.

Be sure that your application is compiled and linked with debugging symbols switched on.

For detecting the running task, the variable “`TCD_Current_Thread`” is used.

To find the OS objects, the labels mentioned in “Manual Configuration” are used.

Features

The OS Awareness for Nucleus supports the following features.

DBUG+ Terminal Emulation

The terminal emulation window can be used to communicate with the target resident Nucleus debugger, called DBUG+. The communication via two memory cells requires no external hardware interface. See the **TERM** command group for a description of the terminal emulation. On request LAUTERBACH can provide you with the source code for the target interface routine.

The 68k example ("~~/demo/m68k/kernel/nucleus/nucleus.cmm") contains this interface and the terminal emulation for the Nucleus debugger DBUG+.

Display of Kernel Resources

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

TASK.TaskStat	Tasks
TASK.MbxStat	Mailboxes
TASK.QueueStat	Queues
TASK.PipeStat	Pipes
TASK.SemaStat	Semaphores
TASK.EventStat	Events
TASK.Timerstat	Timers
TASK.PartMem	Partitions
TASK.DynMem	Dynamic pools
TASK.HIsr	HISRs
TASK.ProcList	Processes

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

Display of History Component

NOTE: This feature is not available on all platforms. If you'd like to use this feature on your platform, but it is not yet supported, please contact LAUTERBACH.

If you added the History Component of Nucleus to your application, the OS Awareness can display the contents of the Nucleus history buffer (which contains kernel calls). See **TASK.HISTory** for details.

Task Stack Coverage

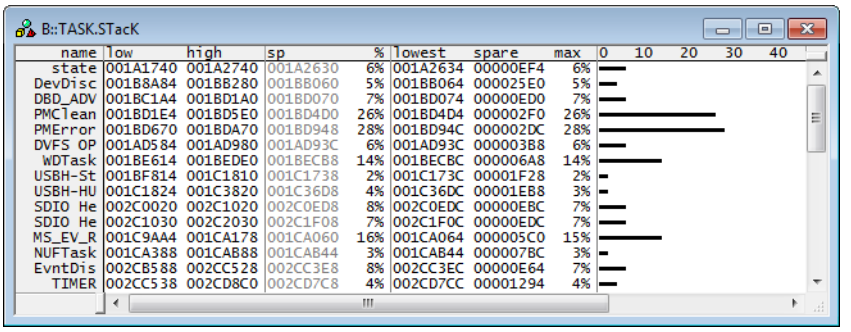
For stack usage coverage of tasks, you can use the **TASK.STacK** command. Without any parameter, this command will open a window displaying with all active tasks. If you specify only a task magic number as parameter, the stack area of this task will be automatically calculated.

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.

To have the task stacks initialized with a pattern, you need to configure the Nucleus kernel accordingly. Depending on the Nucleus version, this may be the configuration option `CFG_NU_OS_KERN_PLUS_CORE_STACK_FILL` or `CFG_NU_OS_KERN_PLUS_COMMON_STACK_FILL`.



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.

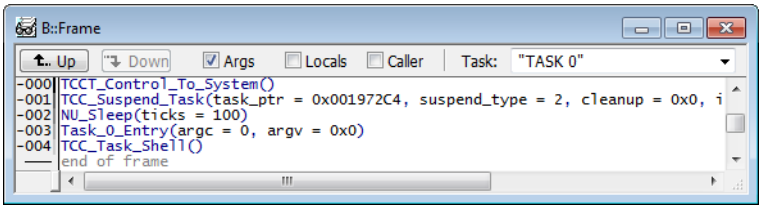
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.



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.

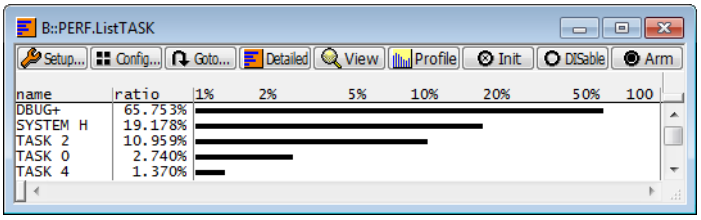
Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

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

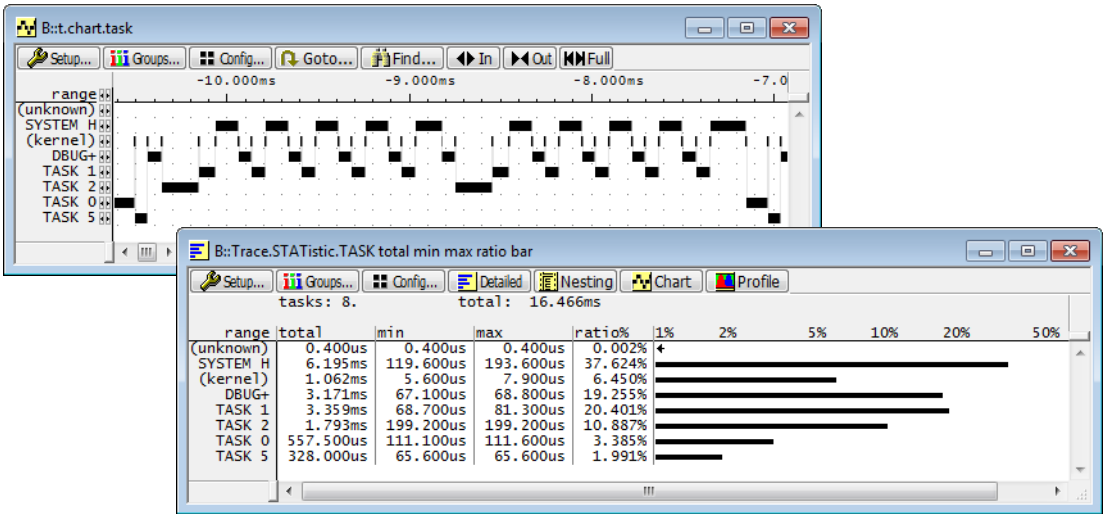
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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



Task State Analysis

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

The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically.

This feature requires that the following data accesses are recorded:

- All accesses to the status words of all tasks
- Accesses to the current task variable (= magic address)

Adjust your trace logic to record all data write accesses, or limit the recorded data to the area where all TCBs are located (plus the current task pointer).

Example: This script assumes that the TCBs are located in an array named TCB_array and consequently limits the tracing to data write accesses on the TCBs and the task switch.

```
Break.Set Var.RANGE(TCB_array) /Write /TraceData
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state timechart

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

All waiting conditions (in Nucleus PLUS called 'suspend') are counted as 'waiting'. The conditions 'finished' and 'terminated' are counted as 'suspended'. The states 'running' and 'ready' calculated as is.

All kernel activities except the scheduler are added to the calling task. The scheduler itself is calculated as "(kernel)".

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:

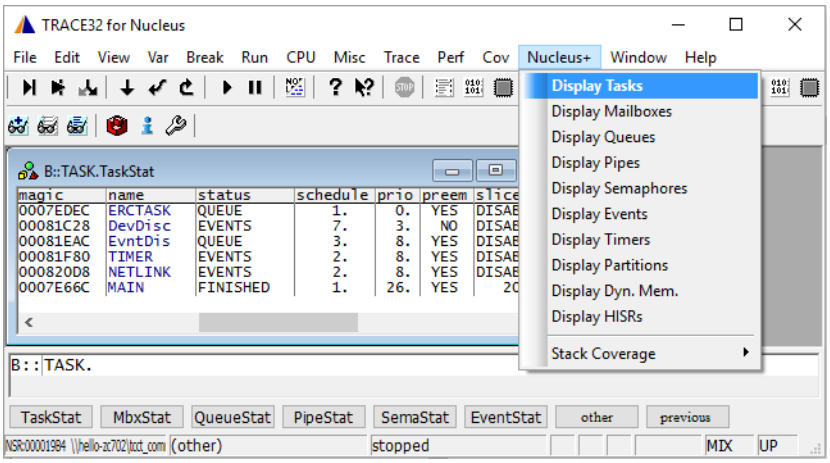
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat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

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

Nucleus specific Menu

The menu file “nucleus.men” contains a menu with Nucleus PLUS specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **Nucleus+**.



- The **DBUG+ Terminal** menu item (if available) brings up a terminal emulation window, which communicates with the preconfigured DBUG+ debugger.
- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the Nucleus specific stack coverage and provides an easy way to add or remove tasks from the stack coverage window.

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

- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with default display.
- The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime statistics or statistics on task states.

Debugging Nucleus Processes

Nucleus can load and link additional software parts, so-called “Processes”. Debugging of these processes is possible with the help of the OS Awareness for Nucleus.

First, load your process into the target. After loading, **TASK.ProcList** should show the newly loaded process. Then, load the symbols of the process using the **Symbol Autoloader**, by either using the command **sYmbol.AutoLOAD.TOUCH**, or by right clicking on the process’ “magic” in **TASK.ProcList** and selecting “Load Symbols”. After this you have access to all symbols of the process. E.g. set a breakpoint on main() and start the process. It should halt at its main entry point.

Symbol Autoloader

The OS Awareness for Nucleus contains an autoloader, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding Nucleus processes 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 reads the target's tables for the processes and fills the autoloader list with the processes found on the target. All necessary information, such as load addresses, are retrieved from kernel-internal information.

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

<action>	Action to take for symbol load, e.g. "`DO ~~/demo/arm/kernel/nucleus/autoload.cmm`"

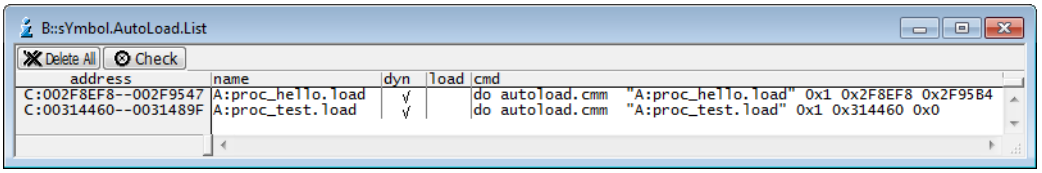
If an address is accessed that is covered by the autoloader list, the autoloader calls *<action>* and appends the load addresses and the space ID of the component to the action. Usually, *<action>* is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example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

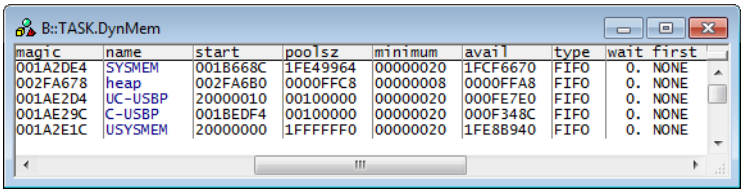
NOTE:	The autoloader covers only components that are already started. Components that are not in the current process table are not covered.


B::sYmbol.AutoLoad.List
Delete All
Check
address
name
dyn
load
cmd
C:002F8EF8--002F9547
A:proc_hello.load
√
do autoload.cmm "A:proc_hello.load" 0x1 0x2F8EF8 0x2F95B4
C:00314460--0031489F
A:proc_test.load
√
do autoload.cmm "A:proc_test.load" 0x1 0x314460 0x0

TASK.DynMem — Display dynamic memory status

Format: **TASK.DynMem** *<dynamic>*

Displays a table with the Nucleus dynamic memory pools. Specifying a dynamic pool magic number will show you the waiting tasks of that dynamic pool.



B::TASK.DynMem

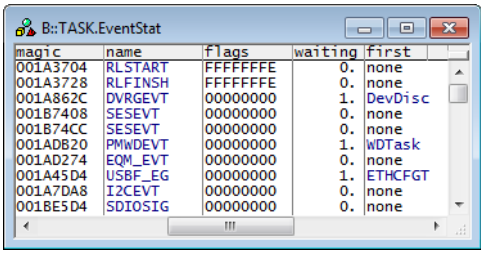
magic	name	start	poolsz	minimum	avail	type	wait	first
001A2DE4	SYSMEM	001B668C	1FE49964	00000020	1FCF6670	FIFO	0.	NONE
002FA678	heap	002FA6B0	0000FFC8	00000008	0000FFA8	FIFO	0.	NONE
001AE2D4	UC-USBP	20000010	00100000	00000020	000FE7E0	FIFO	0.	NONE
001AE29C	C-USBP	001BEDF4	00100000	00000020	000F348C	FIFO	0.	NONE
001A2E1C	USYSMEM	20000000	1FFFFFF0	00000020	1FE8B940	FIFO	0.	NONE

'wait' shows the number of task waiting on this pool.
'first' is the first task waiting.

TASK.EventStat — Display event group status

Format: **TASK.EventStat** *<eventgr>*

Displays a table with the Nucleus event groups. Specifying an event group magic number will show you the waiting tasks of that event group.



B::TASK.EventStat

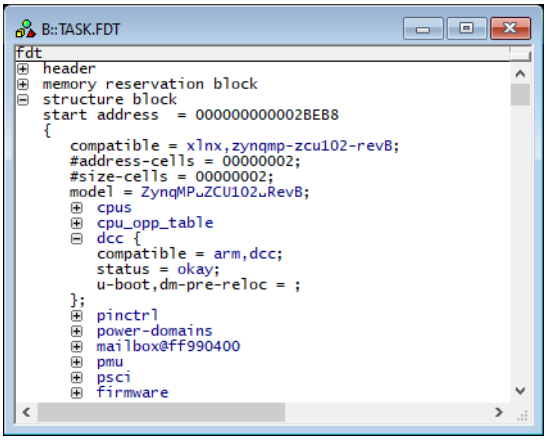
magic	name	flags	waiting	first
001A3704	RLSTART	FFFFFFFE	0.	none
001A3728	RLFINSH	FFFFFFFE	0.	none
001A862C	DVRGEVT	00000000	1.	DevDisc
001B7408	SESEVT	00000000	0.	none
001B74CC	SESEVT	00000000	0.	none
001ADB20	PMWDEVT	00000000	1.	WDTask
001AD274	EQM_EVT	00000000	0.	none
001A45D4	USBF_EG	00000000	1.	ETHCFGT
001A7DA8	I2CEVT	00000000	0.	none
001BE5D4	SDIOSIG	00000000	0.	none

'waiting' shows the number of task waiting on this event group.

'first' is the first task waiting.

Format: **TASK.FDT**

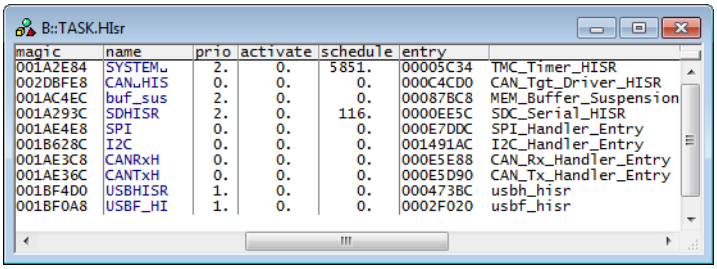
Shows the flattened device tree (aka DTB) used by Nucleus.



TASK.HIsr Display HISRs

Format: **TASK.HIsr** <*hisr*>

Displays a table with the Nucleus HISRs. Specifying a HISR magic number will show you detailed information on that HISR.



B::TASK.HIsr

magic	name	prio	activate	schedule	entry	
001A2E84	SYSTEM␣	2.	0.	5851.	00005C34	TMC_Timer_HISR
002DBFE8	CAN␣HIS	0.	0.	0.	000C4CD0	CAN_Tgt_Driver_HISR
001AC4EC	buf_sus	2.	0.	0.	00087BC8	MEM_Buffer_Suspension
001A293C	SDHISR	2.	0.	116.	0000EE5C	SDC_Serial_HISR
001AE4E8	SPI	0.	0.	0.	000E7DDC	SPI_Handler_Entry
001B628C	I2C	0.	0.	0.	001491AC	I2C_Handler_Entry
001AE3C8	CANRxH	0.	0.	0.	000E5E88	CAN_Rx_Handler_Entry
001AE36C	CANTxH	0.	0.	0.	000E5D90	CAN_Tx_Handler_Entry
001BF4D0	USBHISR	1.	0.	0.	000473BC	usbh_hisr
001BF0A8	USBF_HI	1.	0.	0.	0002F020	usbf_hisr

The fields “magic”, “name” and “entry” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking a HISR magic number will show a local menu.

TASK.HISTory — Display Nucleus history

Format:	**TASK.HISTory**

This feature is not available on all platforms. If you'd like to use this feature on your platform, but it is not yet supported, please contact Lauterbach.

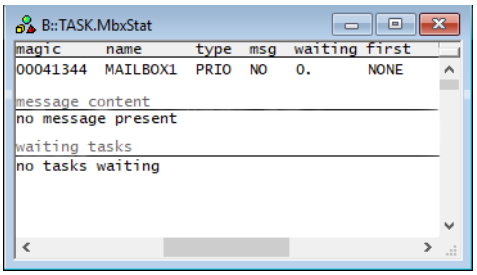
This command shows a window with the content of the Nucleus History Component entries. This is only available, if you included the Nucleus History Component in your application. The display is sorted with showing the oldest entry first and the newest entry last.

Some fields (depending on their meaning) are mouse sensitive. Double click on them to get more information. Double clicking on the entry number will show you the raw data.

TASK.MbxStat — Display mailbox status

Format:	**TASK.MbxStat** <*mailbox*>

Displays a table with the Nucleus mailboxes. Specifying a mailbox magic number will show you the message content and the waiting tasks of that mailbox.

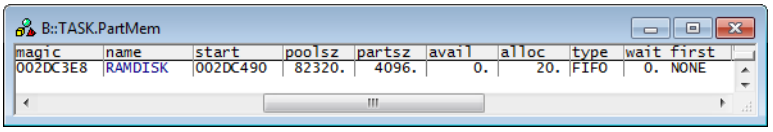


The field 'msg' specifies, whether a message is present or not.
'waiting' shows the number of task waiting on this mailbox.
'first' is the first task waiting.

TASK.PartMem — Display partition memory status

Format:	**TASK.PartMem** *<part>*

Displays a table with the Nucleus partition memory pools. Specifying a partition magic number will show you the waiting tasks of that partition pool.

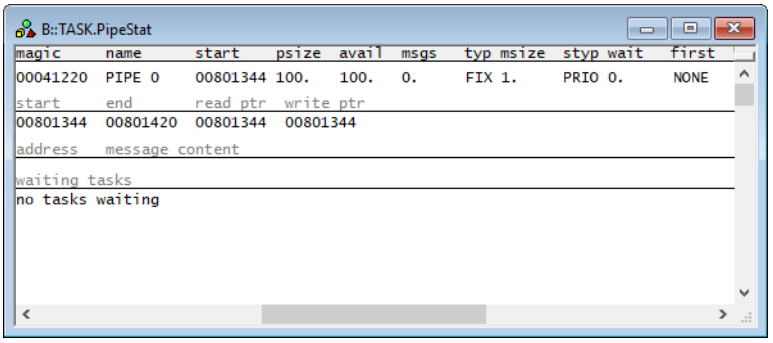


'wait' shows the number of task waiting on this pool.
'first' is the first task waiting.

TASK.PipeStat — Display pipe status

Format:	**TASK.PipeStat** *<pipe>*

Displays a table with the Nucleus pipes. Specifying a pipe magic number will show you the pipe pointers, pipe message contents and the waiting tasks of that pipe.

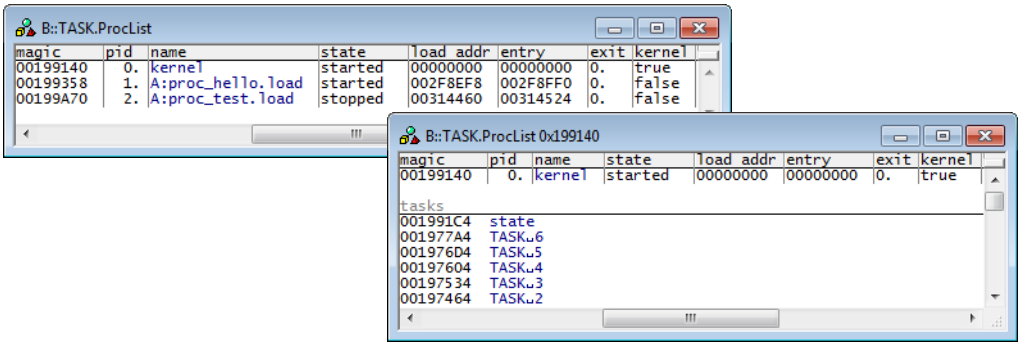


'wait' shows the number of task waiting on this pipe. 'first' is the first task waiting.

TASK.ProcList — Display process list

Format: **TASK.ProcList** *<process>*

Displays a table with all created Nucleus processes. Specifying a process magic number or name will show you detailed information on that process.

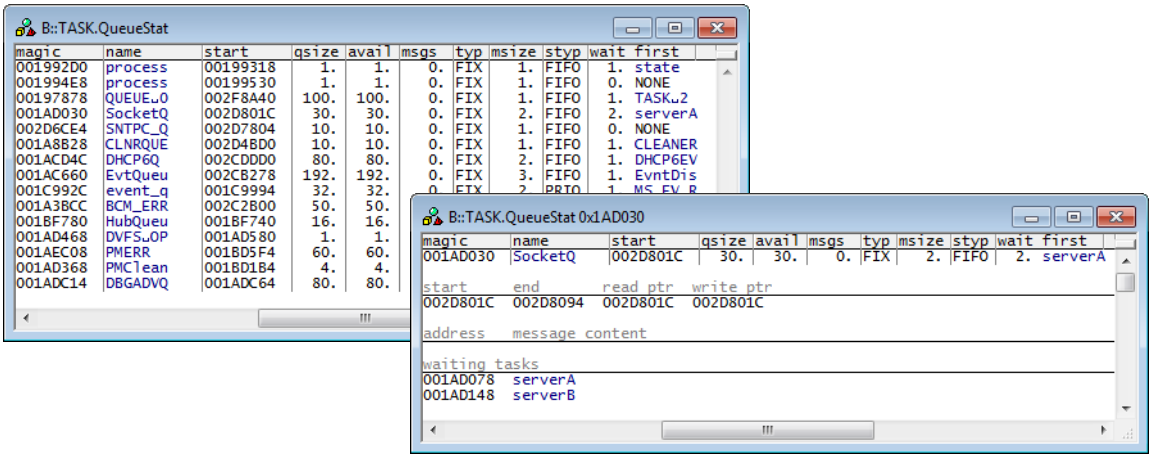


The fields “magic”, “name”, “load addr” and “entry” are mouse sensitive, double clicking on them opens appropriate windows. Right -clicking a process magic number will show a local menu.

TASK.QueueStat — Display queue status

Format: **TASK.QueueStat** *<queue>*

Displays a table with the Nucleus queues. Specifying a queue magic number will show you the queue pointers, queue message contents and the waiting tasks of that queue.

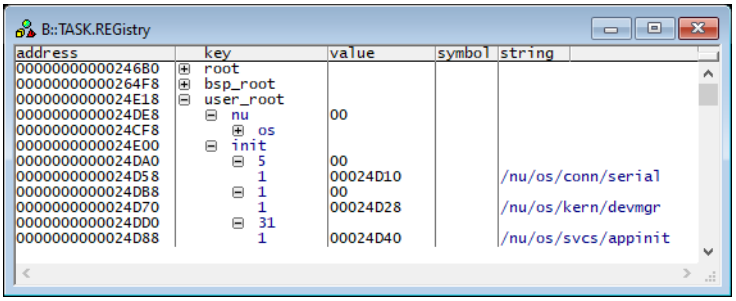


'wait' shows the number of task waiting on this queue. 'first' is the first task waiting.

TASK.REGistry — Display registry entries

Format:	**TASK.REGistry**

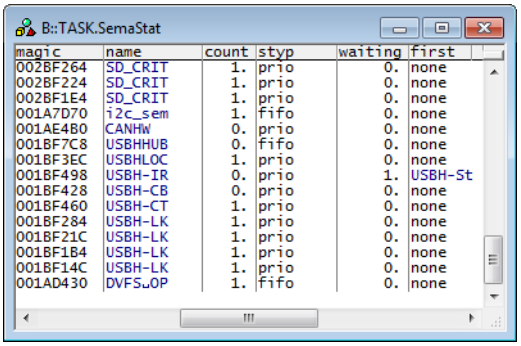
Shows the registry entries of Nucleus.



TASK.SemaStat — Display semaphore status

Format:	**TASK.SemaStat** *<sema>*

Displays a table with the Nucleus semaphores. Specifying a semaphore magic number will show you the waiting tasks of that semaphore.



'waiting' shows the number of task waiting on this semaphore.
'first' is the first task waiting.

TASK.TaskStat — Display task status

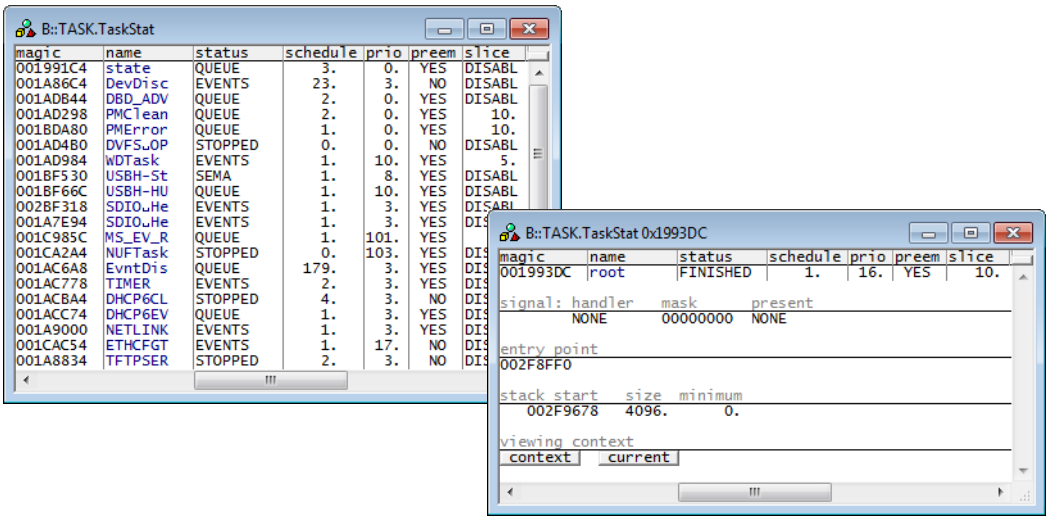
Format:	**TASK.TaskStat** *<task>*

Displays the task table of Nucleus or detailed information about one specific task.

The display is similar to the DBUG+ 'ts' dump.

Without any arguments, a table with all created tasks will be shown.

Specify a task name or magic number to display detailed information on that task.



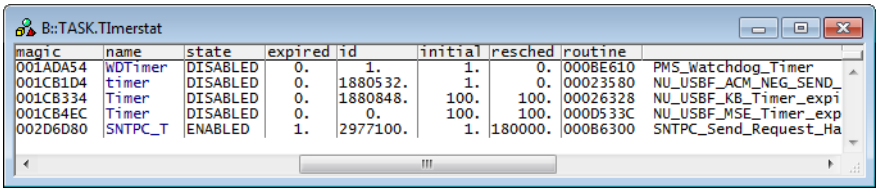
“magic” is a unique ID, used by the OS Awareness to identify a specific task (address of the TCB).

The fields “magic”, “name”, “stkbase” and “handler” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking a value in the **magic** column will show a local menu.

TASK.TImerstat — Display timer status

Format:	**TASK.TImerstat**

Displays a table with the Nucleus application timers.



Nucleus PLUS PRACTICE Functions

There are special definitions for Nucleus PLUS 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.DM.AVAIL() Bytes of dyn. pool

Syntax: **TASK.DM.AVAIL("***<dyn_pool_name>***")**

Returns the available bytes of the specified dyn. pool.

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

Return Value Type: Hex value.

TASK.PL.ENTRY() Entry address of process

Syntax: **TASK.PL.ENTRY(***<process_magic>***)**

Returns the entry address of the specified process.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.