LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual MQX

MANUAL

OS Awareness Manual MQX

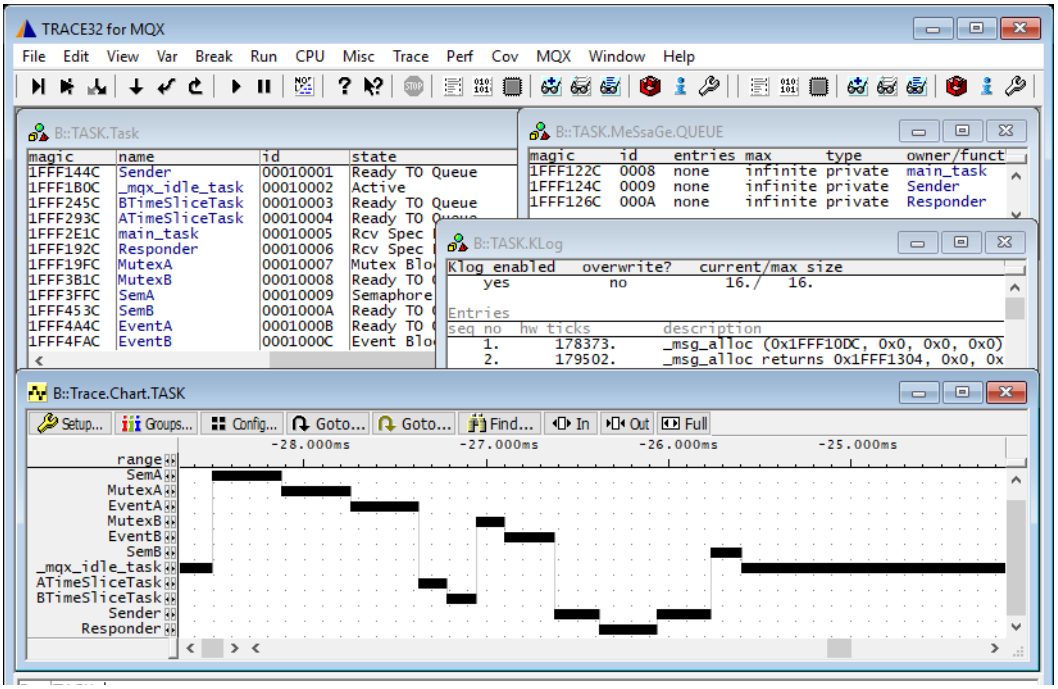
OS Awareness Manual MQX

Version 06-Jun-2024

History

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

Overview



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

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 MQX is supported for the following versions:

- MQX 2.x on 68k, ColdFire, ARM and PowerPC
- MQX 3.6 to MQX 4.0 on ARM, ColdFire and PowerPC.

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

Automatic configuration tries to locate the MQX 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 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:

```
TASK.CONFIG mqx
```

Quick Configuration Guide

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

1. Carefully read the demo startup script (~~/demo/<*processor*>/kernel/mqx/mqx.cmm).
2. Make a copy of the PRACTICE script file “mqx.cmm”.
3. Modify the file according to your application.
4. 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 MQX

No hooks are used in the kernel.

To retrieve all information, the pointers in the initialization structure are used.

Features

The OS Awareness for MQX supports the following features.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. The following information can be displayed:

TASK.TASK	Tasks
TASK.MEMPool	Memory pools
TASK.MeSsaGe.QUEUE	Message queues
TASK.MeSsaGe.POOL	Message pools
TASK.SEMaphore	Semaphores
TASK.EVent	Events
TASK.MuteX	Mutexes
TASK.KLog	Kernel log
TASK.LWSem	Light weight semaphores
TASK.LWEvent	Light weight events
TASK.LWMsgQ	Light weight message queues
TASK.LWMEMPool	Light weight memory pools
TASK.TASKQueue	Task queues

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

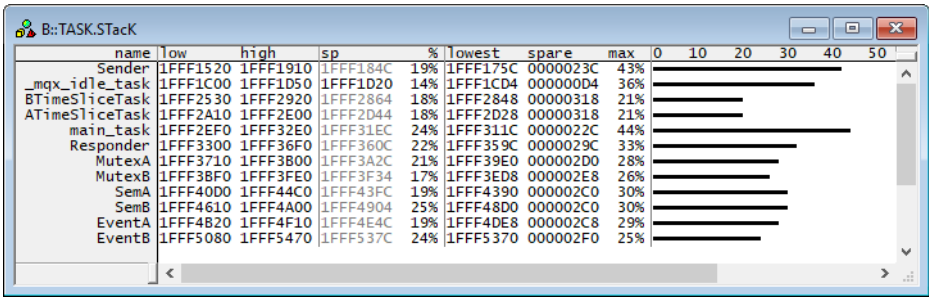
Task Stack Coverage

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-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.

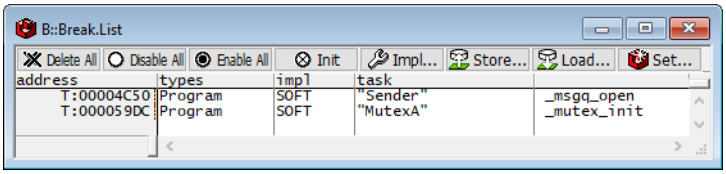
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

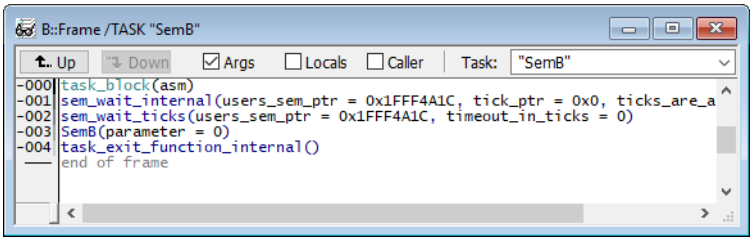
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.

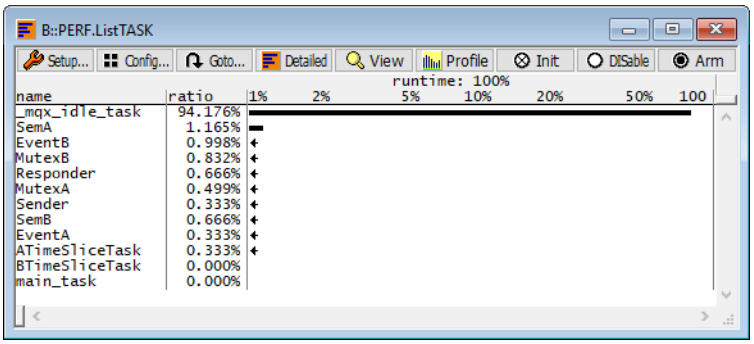


Dynamic Task Performance Measurement

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).



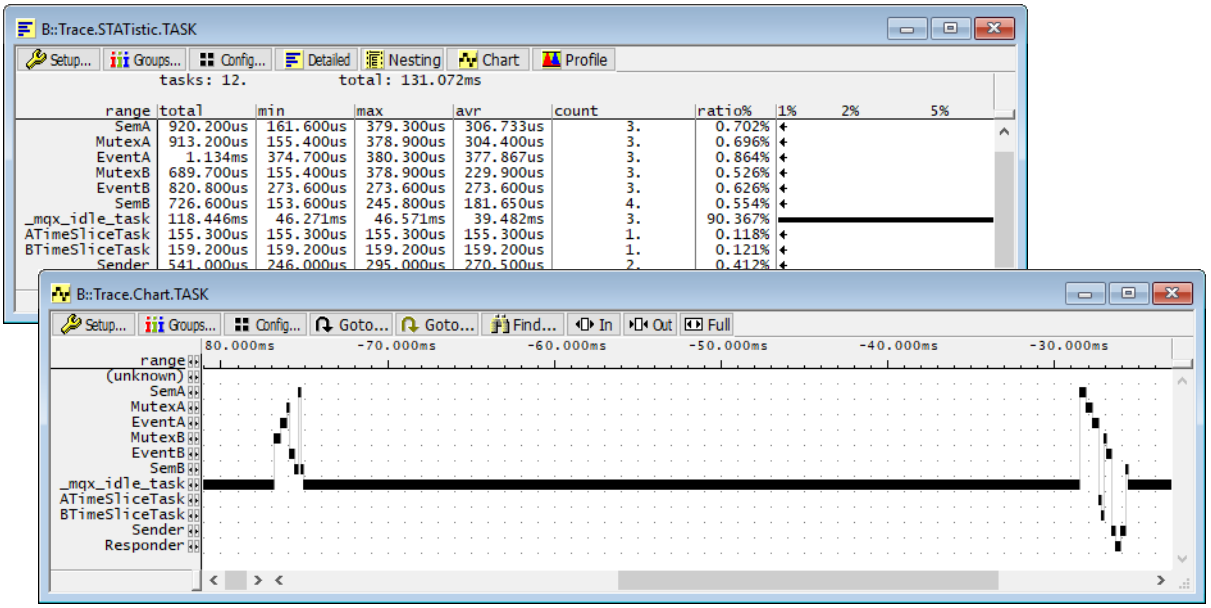
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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

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
```

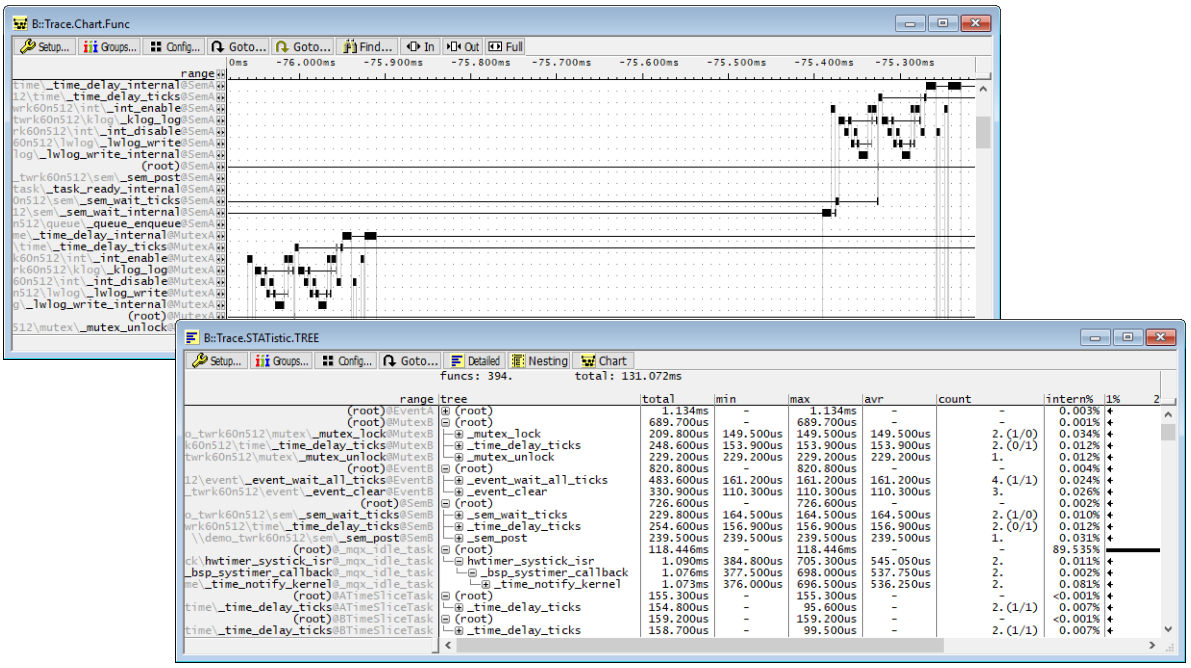
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



MQX specific Menu

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

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

- The **Display** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the MQX specific stack coverage, and provide an easy way to add or remove tasks from the stack coverage window.

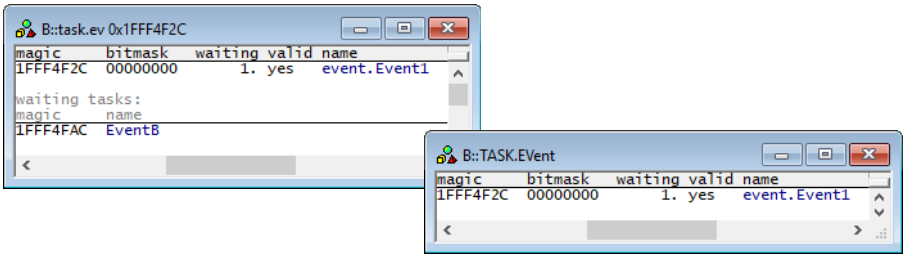
In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- 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 and statistics on task states.

TASK.EVent — Display events

Format: **TASK.EVent** *<event>*

Displays a table with the MQX events. Specifying an event magic number will show you the waiting tasks of that event.



TASK.KLog — Display kernel log

Format: **TASK.KLog**

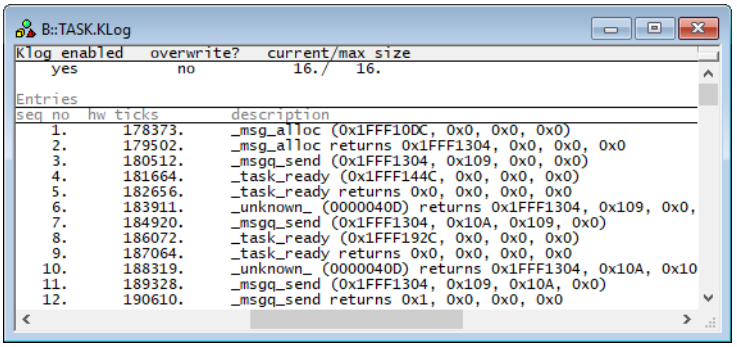
Displays a table with the MQX kernel log.

Each log entry contains a sequence number, a timestamp and information about one of the following entry types:

- Function entry (with up to 4 parameters)
- Function exit (with return value)
- Interrupt (with interrupt number)
- Interrupt end
- Context switch (with name of new task).

The task name field for the context switch is mouse sensitive. Double clicking on it will show the corresponding **TASK.TASK** *<name>* window.

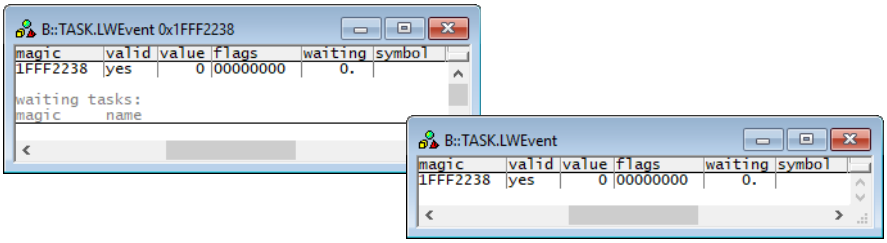
The log entries are displayed in chronological order, with the oldest entry first. If the first entry is not sequence number 1, the oldest entries are being overwritten. The timestamp is obtained from the kernel time when the entry is logged.



TASK.LWEvent Display light weight events

Format: **TASK.LWEvent** *<lwevent>*

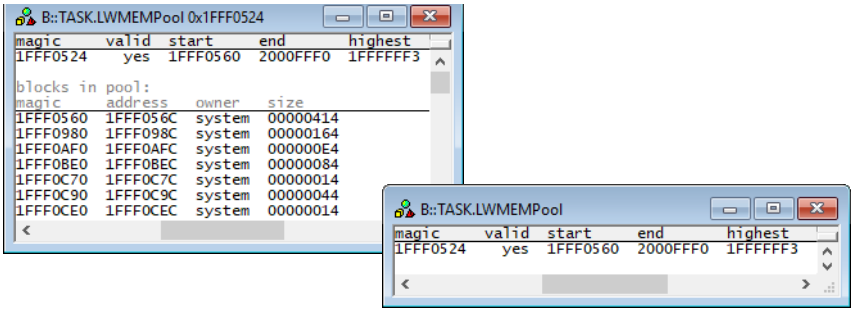
Displays a table with the MQX light weight events. Specifying a light weight event magic number will show you detailed information about the waiting tasks.



TASK.LWMEMPool Display light weight memory pools

Format: **TASK.LWMEMPool** *<lwmempool>*

Displays a table with the MQX light weight memory pools. Specifying a light weight memory pool magic number will show you detailed information about the blocks in the memory pool.



TASK.LWMsgQ Display light weight message queues

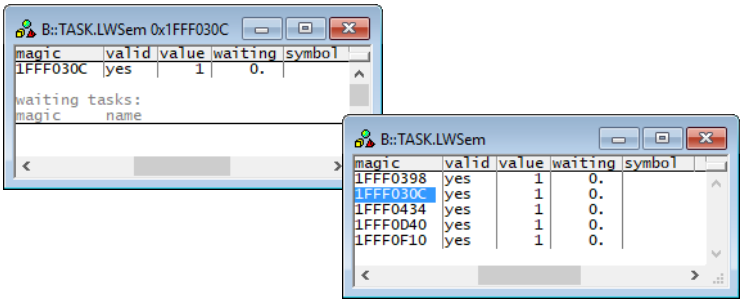
Format: **TASK.LWMsgQ** *<lwmsgq>*

Displays a table with the MQX light weight message queues. Specifying a light weight msg queue magic number will show you detailed information about the waiting tasks.

TASK.LWSem Display light weight semaphores

Format: **TASK.LWSem** *<lwsem>*

Displays a table with the MQX light weight semaphores. Specifying a light weight semaphore magic number will show you detailed information about the waiting tasks.



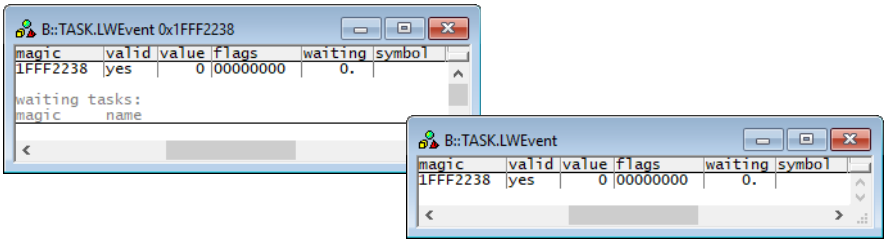
Format: **TASK.MEMPool** *<mempool>*

Displays the task memory pool table of MQX or detailed information about one specific memory pool.

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

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

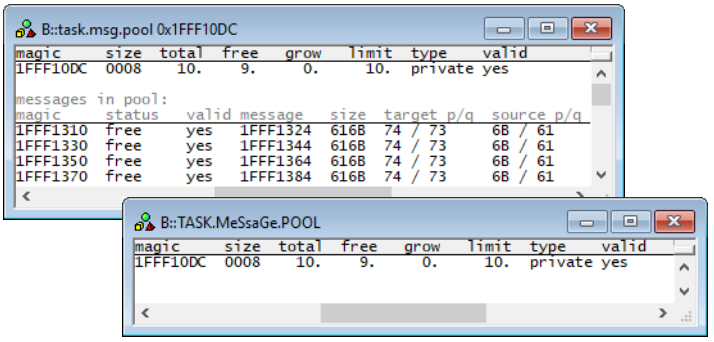
The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.



TASK.MeSsaGe.POOL Display message pools

Format: **TASK.MeSsaGe.POOL** *<msg_p>*

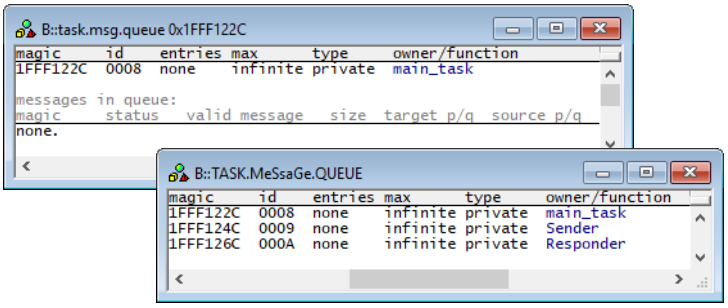
Displays a table with the MQX message pools. Specifying a message pool magic number will show you the messages in that pool.



The fields 'magic' and 'message' are mouse sensitive. Double-clicking on them will perform the appropriate action.

Format: **TASK.MeSsaGe.QUEUE** *<msg_q>*

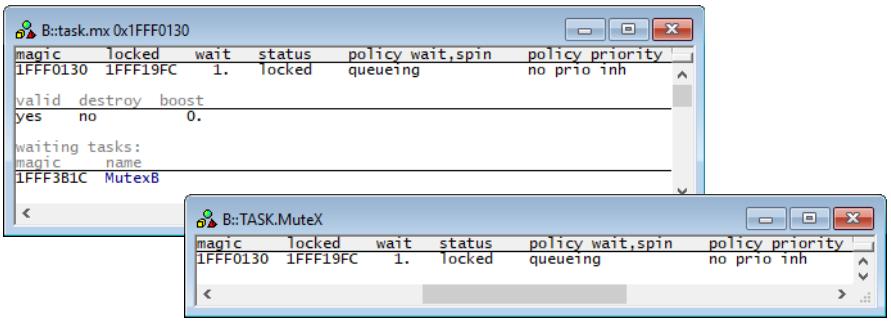
Displays a table with the MQX message queues. Specifying a message queue magic number will show you the messages in that queue.



The fields 'magic' and 'message' are mouse sensitive. Double-clicking on them will perform the appropriate action.

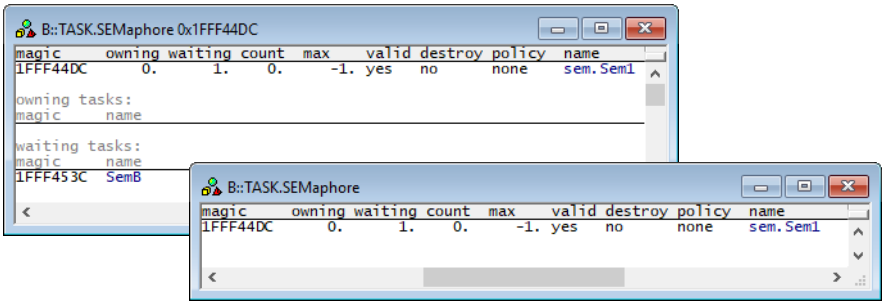
Format: **TASK.MuteX** *<mutex>*

Displays a table with the MQX mutexes. Specifying a mutex magic number will show you detailed information about that mutex.



Format: **TASK.SEMaphore** *<sem>*

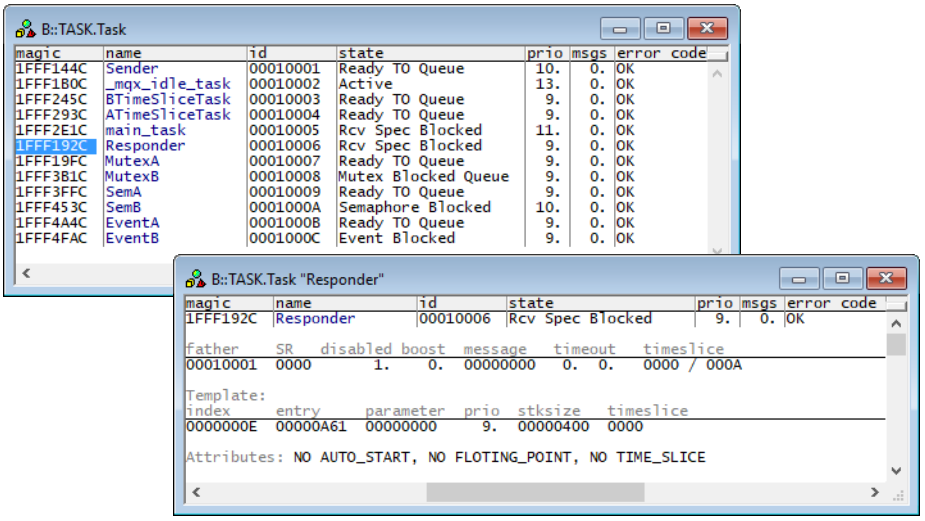
Displays a table with the MQX semaphores. Specifying a semaphore magic number will show you the owning and waiting tasks of that semaphore.



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

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

Without any arguments, a table with all created tasks will be shown.
Specify a task name, ID 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 task object).

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

TASK.TASKQueue Display task queues

Format: **TASK.TASKQueue** *<task_queue>*

Displays a table with the task queues. Specifying a task queue magic number will show you detailed information about this queue.

MQX PRACTICE Functions

There are special definitions for MQX 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.STRUCT() Structure information

Syntax: **TASK.STRUCT(***<item>* | **td)**

Parameter and Description:

<item>	**Parameter Type**: String (***without*** quotation marks). Reports the name of the according structure.
td	**Parameter Type**: String (***without*** quotation marks). Returns the name of the task descriptor structure.

Return Value Type: String.