LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
FreeRTOS

MANUAL

OS Awareness Manual FreeRTOS

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

OS Awareness Manuals ..... 🗁

OS Awareness Manual FreeRTOS **1**

Overview **4**

Brief Overview of Documents for New Users 5

Supported Versions 5

Configuration **6**

Manual Configuration 6

Automatic Configuration 7

Quick Configuration Guide 7

Hooks & Internals in FreeRTOS 8

Features **9**

Display of Kernel Resources 9

Task Stack Coverage 9

Task-Related Breakpoints 11

Task Context Display 12

SMP Support 13

Dynamic Task Performance Measurement 13

Task Runtime Statistics 14

Task State Analysis 15

Function Runtime Statistics 16

FreeRTOS specific Menu 18

FreeRTOS Commands **19**

TASK.EvtGrp Display event groups 19

TASK.MsgBuf Display message buffers 19

TASK.Option Set awareness options 20

TASK.Queue Display queues 20

TASK.Semaphore Display semaphores 21

TASK.StrBuf Display stream buffers 21

TASK.TaskList Display tasks 22

TASK.TImer Display timers 23

FreeRTOS PRACTICE Functions **24**

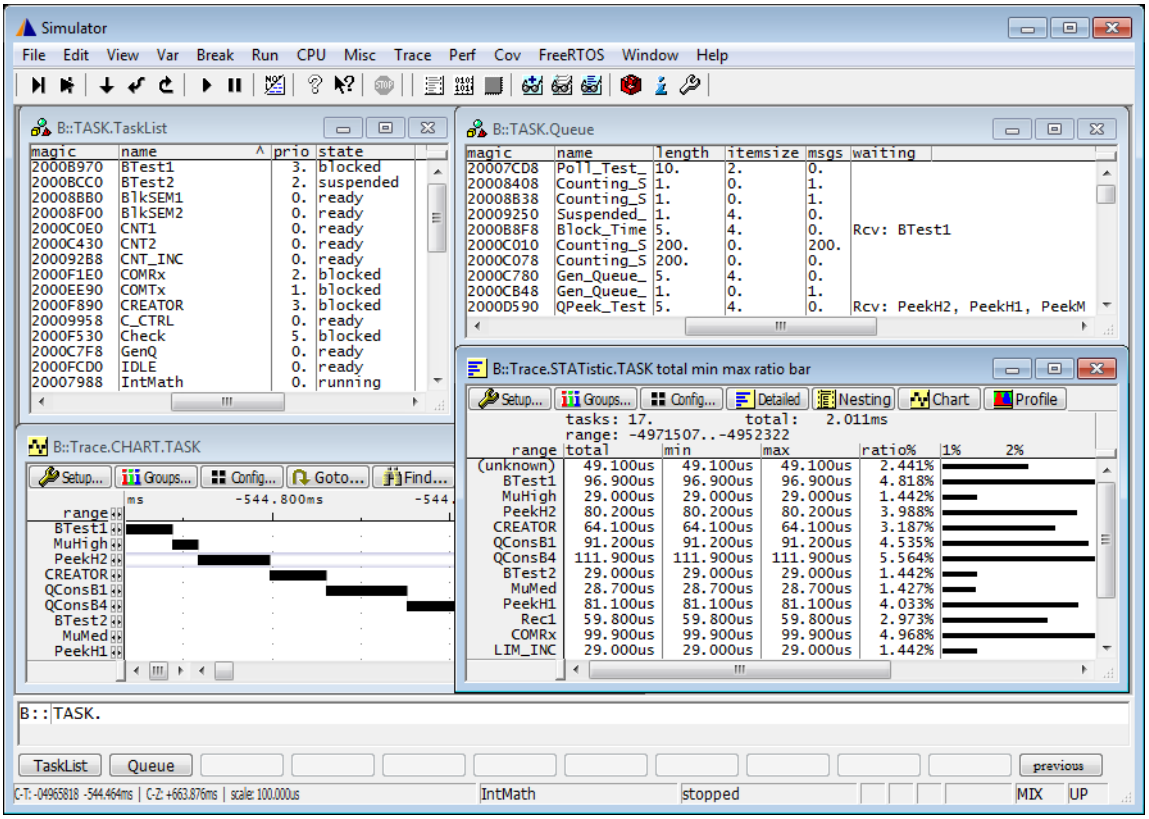
TASK.AVAIL() Availability of FreeRTOS objects 24

TASK.CONFIG() OS Awareness configuration information 24
TASK.STRUCT() Structure names 25

OS Awareness Manual FreeRTOS

Version 06-Jun-2024

Overview



The OS Awareness for FreeRTOS 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 FreeRTOS is supported for the following version:

- FreeRTOS V4.x to V11.x on ARC, ARM, ARM64, AVR32, Beyond, ColdFire, H8S, HC12, MicroBlaze, MIPS, Nios II, PowerPC, STRed, TMS320C2/6/7xxx, TriCore, and Xtensa.
- SafeRTOS V5.x to V9.x on ARM, PowerPC, TMS320C6/7xxx and TriCore

Configuration

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

Automatic configuration tries to locate the FreeRTOS internals automatically. For this purpose all symbol tables have to 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 has to 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 FreeRTOS can be used to explicitly define some operational values.

Format: **TASK.CONFIG ~~/demo/*<arch>*/kernel/freertos/freertos.t32**
<magic_address> *<stack_size>*

<magic_address> Specifies a memory location that contains the current running task. This address can be found at "`pxCurrentTCB`". Either use this label or specify 0 to detect it automatically.

<stack_size> Some FreeRTOS versions do not provide the stack size in a running system. To do a **stack coverage analysis**, the debugger needs to know the stack size. In this case, specify the stack size in bytes as second parameter. Calculate it by
`configMINIMAL_STACK_SIZE * sizeof(portSTACK_TYPE)`
(see your FreeRTOSConfig.h file). If your FreeRTOS version provides the stack size, use **automatic configuration** instead.

The stack size can also be set using the command **TASK.Option STacKSIZE**.

Example:

```
; application uses 256 words for stack size:
TASK.CONFIG freertos.t32 0 256.*4
```

See **Hooks & Internals** for details.

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:

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

If a system symbol is not available or if another address should be used for a specific system variable, or if your FreeRTOS version doesn't provide the stack sizes of the tasks, then the corresponding argument has to be set manually with the appropriate value (see **'Manual Configuration'**).

See also the example "~~/demo/<*arch*>/kernel/freertos/freertos.cmm".

Refer to **'Hooks & Internals'** for details on the used symbols.

Quick Configuration Guide

To get a quick access to the features of the OS Awareness for FreeRTOS 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/freertos/freertos.t32
   ```

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

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

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

Now you can access the FreeRTOS extensions through the menu.

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

No hooks are used in the kernel.

For detecting the current running task, the kernel symbol 'pxCurrentTCB' is used.

For retrieving the kernel data and structures, the OS Awareness uses the global kernel symbols and structure definitions. Ensure that access to those structures is possible every time when features of the OS Awareness are used.

For automatic detection of stack sizes, the OS Awareness uses either the "usStackDepth" or the "pxEndOfStack" member variable of the "tskTCB" structure. When using FreeRTOS version 10 or above, set configRECORD_STACK_HIGH_ADDRESS to 1 to get a full stack coverage. If automatic detection of stack sizes is available, use **Automatic configuration**. If it is not available, **TASK.Option STacKSIZE** or use **Manual configuration** and provide the stack size manually.

FreeRTOS allows queues and semaphores to be "registered". If you configured FreeRTOS co contain a queue registry (configQUEUE_REGISTRY_SIZE), **TASK.Queue** and **TASK.Semaphore** without parameters will show all queues registered with vQueueAddToRegistry(). Otherwise you have to specify a queue or semaphore handle as parameter.

Features

The OS Awareness for FreeRTOS supports the following features.

Display of Kernel Resources

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

TASK.TaskList	Tasks
TASK.Queue	Queues
TASK.Semaphore	Semaphores
TASK.TImer	Timers
TASK.EvtGrp	Event Groups
TASK.StrBuf	Stream Buffers
TASK.MsgBuf	Message Buffers

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

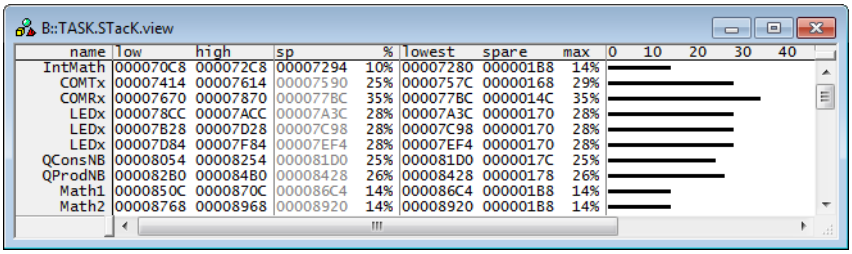
Task Stack Coverage

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: When using a FreeRTOS version 10 and above, configure your system with

```
#define configRECORD_STACK_HIGH_ADDRESS 1
#define configCHECK_FOR_STACK_OVERFLOW  2
```

to get a full stack coverage. Use **TASK.STacK.PATtern** to set the stack fill pattern as defined in task.c: `tskSTACK_FILL_BYTE`.

When using a FreeRTOS version 5 to 9, FreeRTOS does not provide information about the stack sizes. You need to specify the stack size in the configuration of the OS Awareness. See **Hooks & Internals** and **Manual Configuration** for details.

The manual configuration only allows to set one stack size for all tasks (usually the minimal stack size). If you want to override the stack characteristics of one task, you can use a small script to do so.
Example to set the stack size of the “IDLE” task to 1024 bytes:

```
; Adapt stack characteristics of a task
; Specify the task name, e.g. the IDLE task:
&task="IDLE"
; Specify the new task size in bytes for this task, e.g. 1024 bytes:
&stacksize=0x400
; Open standard stack view and ensure a display update
TASK.STacK.view
SCREEN
; Calculate task “magic” and stack start address
&magic=task.magic("IDLE")
&stackstart=var.value(((tskTCB*)&magic)->pxStack)
; Remove the standard stack calculation for this task
TASK.STacK.ReMove &magic
; And add the custom one:
TASK.STacK.ADD &magic &stackstart++(&stacksize-1)
```

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.

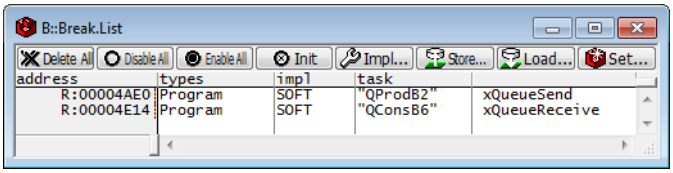
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

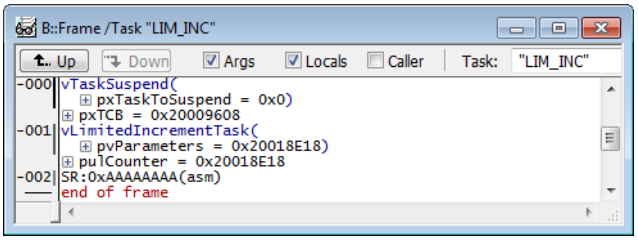
- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

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.

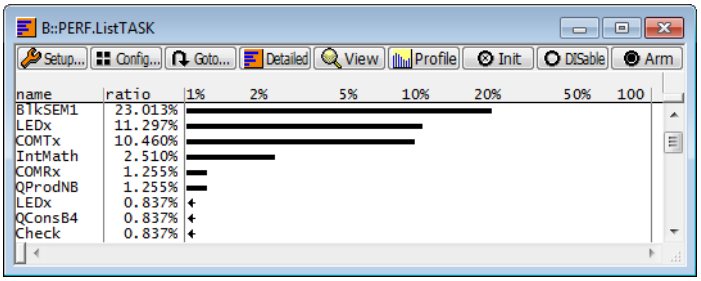
In SMP systems, the **TASK.TaskList** command shows at which core a task is running, if it is in the running state.

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

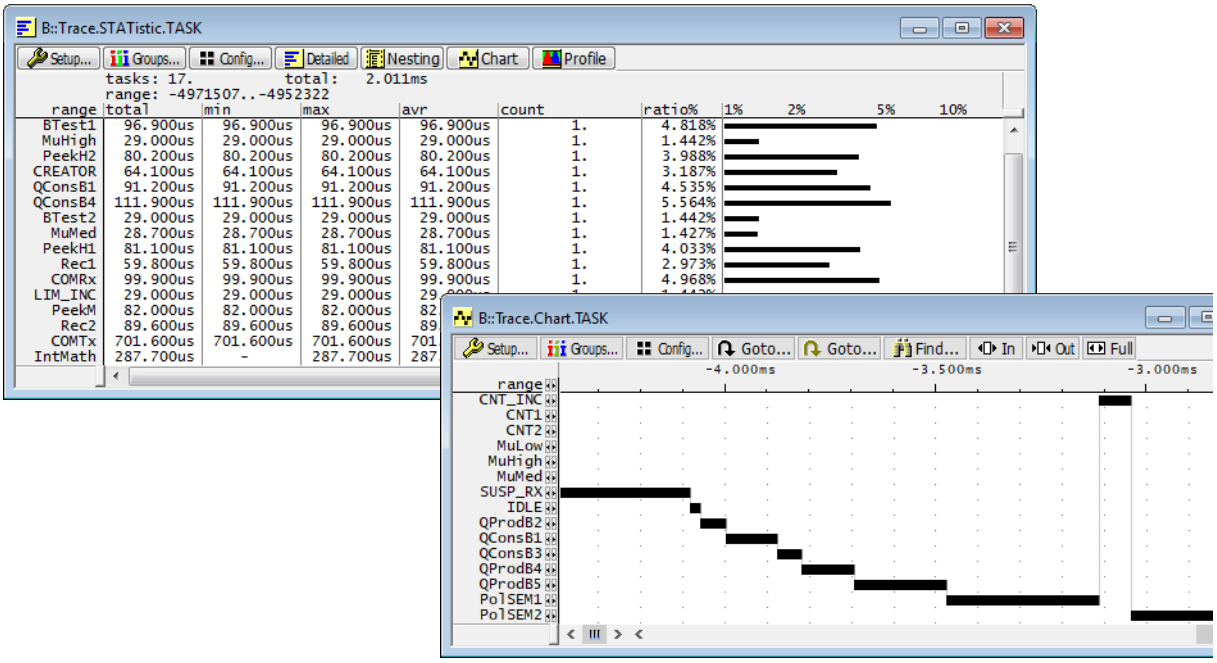
NOTE:	This feature is *only* available, if your debug environment is available to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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

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
```

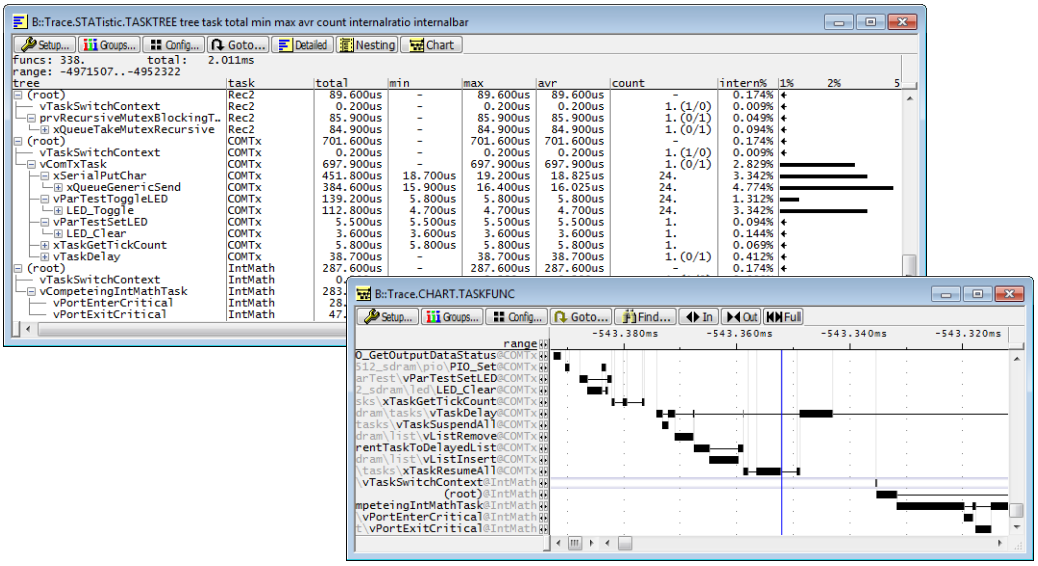
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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

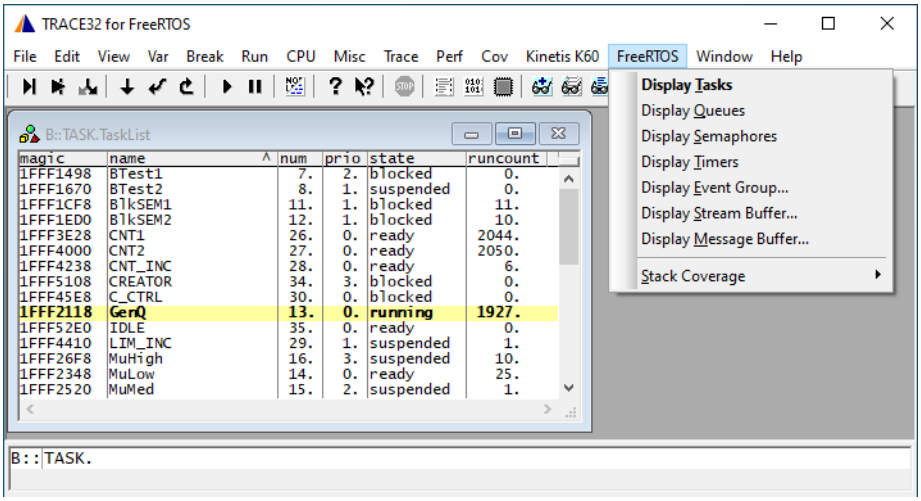
The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".



FreeRTOS specific Menu

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

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



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

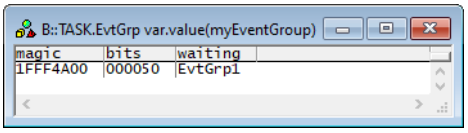
In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- 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 and statistics on task states.

TASK.EvtGrp — Display event groups

Format: **TASK.EvtGrp** <*evtgrp*>

Displays detailed information about one specific event group. Specify an event group handle as parameter.



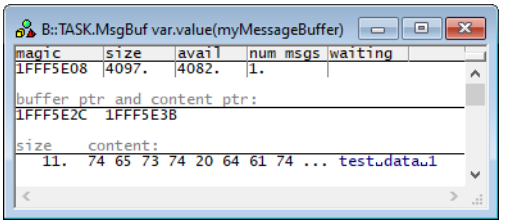
'magic' is a unique ID, used by the OS Awareness to identify a specific event group (address of the EventGroup_t structure).

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

TASK.MsgBuf — Display message buffers

Format: **TASK.MsgBuf** <*msgbuf*>

Displays detailed information about one specific message buffer. Specify a message buffer handle as parameter.



'magic' is a unique ID, used by the OS Awareness to identify a specific message buffer (address of the StreamBuffer_t structure).

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

Format:	**TASK.Option** <*option*>
<*option*>:	**STacKSIZE** <*size*>

Sets options to the awareness.

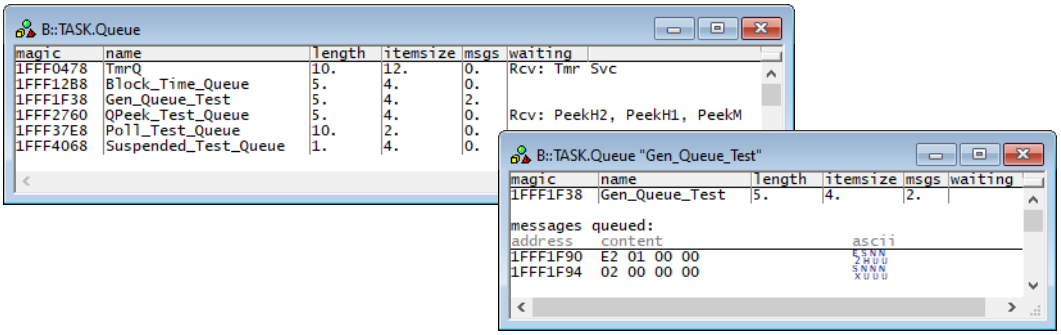
STacKSIZE <*size*>
: Some FreeRTOS versions do not provide the stack size in a running system.
To do a stack coverage analysis, the debugger needs to know the stack size. In this case, specify the stack size in bytes as second parameter.
Calculate it by
`configMINIMAL_STACK_SIZE * sizeof(portSTACK_TYPE)`
(see your FreeRTOSConfig.h file)
See **Hooks & Internals** for details.

TASK.Queue Display queues

Format:	**TASK.Queue** [<*queue*>]

Displays the registered queue table or detailed information about one specific queue.

FreeRTOS allows queues to be “registered”. If you configured FreeRTOS to contain a queue registry (`configQUEUE_REGISTRY_SIZE`), TASK.Queue without parameters will show all queues registered with vQueueAddToRegistry(). Otherwise you have to specify a queue handle as parameter, to display information on that queue.



’magic’ is a unique ID, used by the OS Awareness to identify a specific queue (address of the xQUEUE object).

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

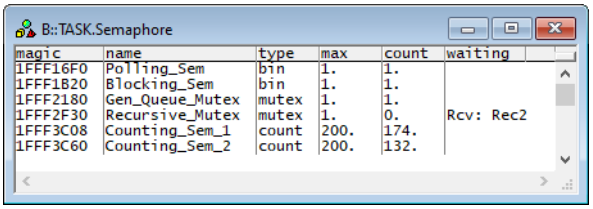
Note: “Queue Sets” in FreeRTOS are internally organized as normal queues. There is no way to detect a queue set as such.

TASK.Semaphore — Display semaphores

Format: **TASK.Semaphore** [<*semaphore*>]

Displays the registered semaphore table or detailed information about one specific semaphore.

FreeRTOS allows semaphores to be “registered”. If you configured FreeRTOS to contain a queue registry (`configQUEUE_REGISTRY_SIZE`), TASK.Semaphore without parameters will show all semaphores registered with vQueueAddToRegistry(). Otherwise you have to specify a semaphore handle as parameter, to display information on that semaphore.



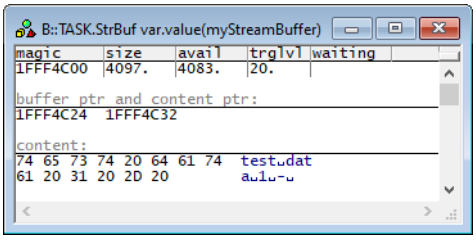
’magic’ is a unique ID, used by the OS Awareness to identify a specific semaphore (address of the xQUEUE object).

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

TASK.StrBuf — Display stream buffers

Format: **TASK.StrBuf** <*strbuf*>

Displays detailed information about one specific stream buffer. Specify a stream buffer handle as parameter.



’magic’ is a unique ID, used by the OS Awareness to identify a specific stream buffer (address of the StreamBuffer_t structure).

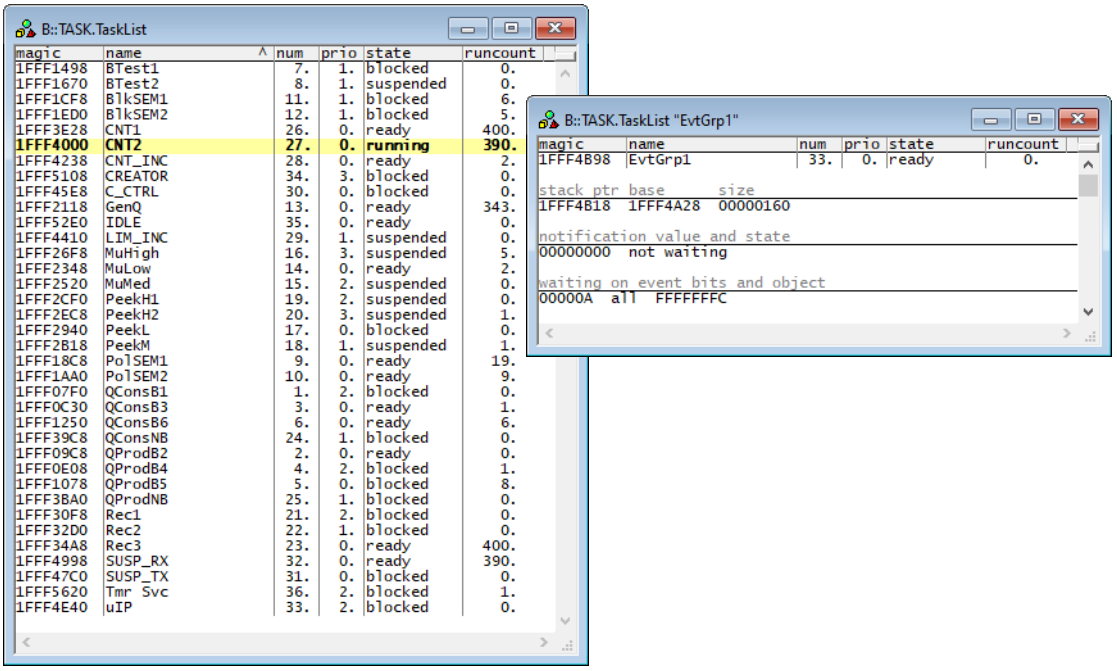
The field 'magic' is mouse sensitive, double clicking on it opens appropriate windows. Right clicking on it will show a local menu.

TASK.TaskList — Display tasks

Format: **TASK.TaskList** [<*task*>]

Displays the task table of FreeRTOS or detailed information about one specific task.
The display is similar to the FreeRTOS API function 'vTaskList()'.

TASK.TaskList without parameters will show all tasks. Specify a task name in quotes, or a task magic to see detailed information about this task.



You can sort the window to the entries of a column by clicking on the column header.

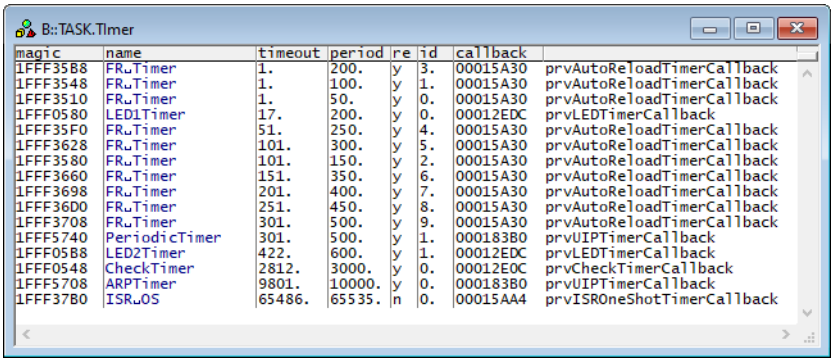
'magic' is a unique ID, used by the OS Awareness to identify a specific task (address of the TCB).

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

Format: **TASK.TImer** [<*timer*>]

Displays the software timer table or detailed information about one specific timer.

TASK.TImer without parameters will show all software timers created, Specify a timer handle as parameter to display information on that timer.



magic	name	timeout	period	re	id	callback	
1FFF35B8	FR_Timer	1.	200.	y	3.	00015A30	prvAutoReloadTimerCallback
1FFF3548	FR_Timer	1.	100.	y	1.	00015A30	prvAutoReloadTimerCallback
1FFF3510	FR_Timer	1.	50.	y	0.	00015A30	prvAutoReloadTimerCallback
1FFF0580	LED1Timer	17.	200.	y	0.	00012EDC	prvLEDTimerCallback
1FFF35F0	FR_Timer	51.	250.	y	4.	00015A30	prvAutoReloadTimerCallback
1FFF3628	FR_Timer	101.	300.	y	5.	00015A30	prvAutoReloadTimerCallback
1FFF3580	FR_Timer	101.	150.	y	2.	00015A30	prvAutoReloadTimerCallback
1FFF3660	FR_Timer	151.	350.	y	6.	00015A30	prvAutoReloadTimerCallback
1FFF3698	FR_Timer	201.	400.	y	7.	00015A30	prvAutoReloadTimerCallback
1FFF36D0	FR_Timer	251.	450.	y	8.	00015A30	prvAutoReloadTimerCallback
1FFF3708	FR_Timer	301.	500.	y	9.	00015A30	prvAutoReloadTimerCallback
1FFF5740	PeriodicTimer	301.	500.	y	1.	000183B0	prvUIPTimerCallback
1FFF05B8	LED2Timer	422.	600.	y	1.	00012EDC	prvLEDTimerCallback
1FFF0548	CheckTimer	2812.	3000.	y	0.	00012E0C	prvCheckTimerCallback
1FFF5708	ARPTimer	9801.	10000.	y	0.	000183B0	prvUIPTimerCallback
1FFF37B0	ISR_OS	65486.	65535.	n	0.	00015AA4	prvISROneShotTimerCallback

'magic' is a unique ID, used by the OS Awareness to identify a specific timer (address of the xTIMER object).

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

FreeRTOS PRACTICE Functions

There are special definitions for FreeRTOS specific PRACTICE functions.

TASK.AVAIL() Availability of FreeRTOS objects

Syntax: **TASK.AVAIL(qreg)**

Reports the availability of FreeRTOS objects.

Parameter and Description:

qreg	**Parameter Type**: String (***without*** quotation marks). Returns 1 if FreeRTOS has a queue registry.

Return Value Type: Hex value.

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.

Syntax: **TASK.STRUCT(queue | tcb | timer)**

Reports the structure names of FreeRTOS objects.

Parameter and Description:

queue	**Parameter Type**: String (***without*** quotation marks). Returns the structure name of queues.
tcb	**Parameter Type**: String (***without*** quotation marks). Returns the structure name of the TCB.
timer	**Parameter Type**: String (***without*** quotation marks). Returns the structure name of software timers.

Return Value Type: Hex value.