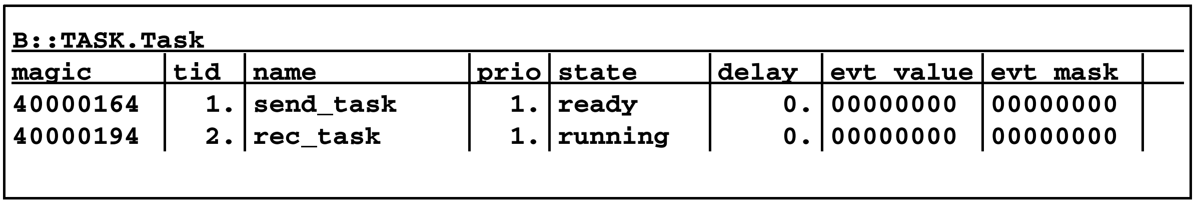
RTOS Debugger for RTX-ARM

Version 02-Apr-2014

Overview



```
B::TASK.Task
magic    |tid|name          |prio|state    |delay|evt value|evt mask|
40000164 | 1.|send_task     |  1.|ready    |   0.|00000000 |00000000|
40000194 | 2.|rec_task      |  1.|running  |   0.|00000000 |00000000|
```

The RTOS Debugger for RTX-ARM 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:

- **”Debugger Basics - Training”** (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**.
- **“RTOS Debugger”** (rtos_<x>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate RTOS manual informs you how to enable the OS-aware debugging.

Supported Versions

Currently RTX-ARM is supported for the following versions:

- RTX-ARM V3.x and V4.x on all ARM derivatives

Configuration

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

Automatic configuration tries to locate the RTX-ARM internals automatically. For this purpose all symbol tables must be loaded and accessable at any time the RTOS debugger 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 do use the display functions “On The Fly”, i.e. displaying the OS objects, while the target is running, you need to have access to memory while running. In case of a ICE or FIRE, you have to map emulation or shadow memory to the address space of all used system tables. In case of ICD, you have to enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent).

Manual Configuration

Manual configuration is not necessary for the RTX-ARM RTOS debugger.

Automatic Configuration

For system resource display and trace functionality, you can do an automatic configuration of the RTOS debugger. For this purpose it is neccessary, that all system internal symbols are loaded and accessable at any time, the RTOS debugger 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 full automatic configuration omit all arguments:

Format:	**TASK.CONFIG rtx**

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 also “**Hooks & Internals**” for details on the used symbols.

Quick Configuration Guide

To get a quick access to the features of the RTX-ARM RTOS debugger with your application, follow the following roadmap:

1. Copy the files "`rtx.t32`" and "`rtx.men`" to your project directory (from TRACE32 directory "demo/<*processor*>/kernel/rtx").
2. Start the TRACE32 Debugger.
3. Load your application as normal.
4. Execute the command "`TASK.CONFIG rtx`" (See "**Automatic Configuration**").
5. Execute the command "`MENU.ReProgram rtx`" (See "**RTX-ARM Specific Menu**").
6. Start your application.

Now you can access the RTX-ARM extensions through the menu.

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

Hooks & Internals in RTX-ARM

No hooks are used in the kernel.

For detecting the current running task, the kernel symbol "`os_runtask`" is used.

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

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

Features

The RTOS debugger for RTX-ARM supports the following features.

Display of Kernel Resources

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

- tasks	**TASK.Task**

For a detailed description of each command refer to the chapter “**RTX-ARM Commands**”.

If your hardware allows accessing the memory, 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 RTX-ARM tasks, you can use the **TASK.STacK** command. Without any parameter, this command will set up a window with all active RTX-ARM tasks. If you specify only a magic number as parameter, the stack area of this task will be automatically caculated.

To use the calculation of the maximum stack usage, flag memory must be mapped to the task stack areas, when working with the emulation memory. When working with the target memory, 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** rsp. **TASK.STacK.ReMove** commands with the task magic number as parameter, or omit the parameter and select from the task list window.

It is recommended, to display only the tasks, that 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:

Format:	**Break.Set** *<address>*|*<range>* [*</option>*] **/TASK** *<task>*

Use a task magic, id or name for <task>.

This task related breakpoint is implemented by a conditional breakpoint inside the debugger. I.e., 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.

Please note, that this feature impacts the real-time behavior of the application.

For a general description of the **Break.Set** command, please see its documentation.

When single stepping, the debugger halts on the next instruction, regardless which task hits this breakpoint. When debugging shared code, stepping over an OS function may lead to a task switch and coming back to the same place - but with a different task. If you want to “stick” the debugging within 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 events.

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.

Dynamic Task Performance Measurement

The debugger may execute a dynamic performance measurement by evaluating the current running task in changing time intervals. Start the measurement with the command **PERF.Mode TASK**, 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 OSes (SYStem.Option MMUSPACES ON), then you need to set PERF.MMUSPACES, too.

For a general description of the **PERF** command, refer to **”General Commands Reference Guide P”** (general_ref_p.pdf).

Task Runtime Statistics

NOTE: This feature is **only** available, if your debugger equipment is able to trace task switches (see below).

Out of the recordings done by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically. To do this, the debugger must be able to detect the task switches out of the trace, i.e. the task switches need to be recorded.

Usually, there's a variable in the system that holds the current running task. By recording the accesses to this memory location (aka "magic" location), the debugger detects a task switch by a change of this variable. Please note, that the debugger equipment must be able to trace memory data accesses in this case, program flow trace is not sufficient.

If a hardware trace solution is available, that provides a so called "context id" (e.g. ARM11 ETM), and if this context id is served by the operating system, it is sufficient to enable the recording of the context id (owner cycle); no data trace is needed.

To do a selective recording on task switches with state analyzers (ICE and FIRE), use the following PRACTICE commands:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Program the Analyzer to record only task switches
Analyzer.ReProgram
(
    Sample.Enable if AlphaBreak&&Write
)
```

To do a selective recording on task switches with flow traces (ICD, e.g. ETM and NEXUS trace), based on the data accesses, use the following PRACTICE command:

```
; Enable tracing only on the magic location
Break.Set task.config(magic) /TraceEnable
```

To do a selective recording on task switches with flow traces, based on the context id, use the following PRACTICE command:

```
; Enable tracing of the context id (e.g. 32bit)
ETM.ContextID 32
```

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 “(root)”.

Task State Analysis

NOTE:	This feature is **only** available, if your debugger equipment is able to trace memory data accesses (program flow trace is not sufficient).

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

This feature needs recording of all accesses to the status words of all tasks. Additionally, the accesses to the current task pointer (=magic) are needed. 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).

To do a selective recording on task states with state analyzers (ICE or FIRE), use **TASK.TASKState**, if available, to mark the status words with Alpha breakpoints. Run the following PRACTICE script:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Mark all task state words with Alpha breakpoints
TASK.TASKState

; Program the Analyzer to record task state transitions
Analyzer.ReProgram
(
    Sample.Enable if AlphaBreak&&Write
)
```

To do a selective recording on task states with flow traces (ICD, e.g. ETM and NEXUS trace), just enable the recording of all data write cycles.

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 “(root)”.

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 debugger equipment is able to trace memory data accesses (program flow trace is not sufficient).

All function related statistic and timechart 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, additionally to the function entries and exits, the task switches must be recorded.

To do a selective recording on task related function runtimes with state analyzers (ICE and FIRE), use the following PRACTICE commands:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Mark the function entries/exits with Alpha/Beta breakpoints
Break.SetFunc

; Program the Analyzer to record function entries/exits and task switches
Analyzer.ReProgram
(
    Sample.Enable if AlphaBreak||BetaBreak
    Mark.A        if AlphaBreak
    Mark.B        if BetaBreak
)
```

To do a selective recording on task related function runtimes with flow traces (ICD, e.g. ETM and NEXUS trace), use the following PRACTICE command:

```
; Enable tracing only on the magic location
Break.Set task.config(magic) /TraceData
```

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

Trace.List List.TASK FUNC	Display function nesting
Trace.STATistic.TASKFunc	Display function runtime statistic
Trace.STATistic.TASKTREE	Display functions as call tree
Trace.Chart.TASKFunc	Display function timechart

The start of the recording time, when the calculation doesn’t know, which task is running, is calculated as “(root)”.

RTX-ARM Specific Menu

The file “rtx.men” contains an alternate menu with RTX-ARM specific topics. Load this menu with the **MENU.ReProgram** command.

You will find a new pull-down menu called “RTX-ARM”.

The “Display” topics launch the kernel resource display windows.

The “Stack Coverage” submenu starts and resets the RTX-ARM specific stack coverage, and provides an easy way to add or remove tasks from the stack coverage window.

The “Trace” pull-down menu is extended. In the “List” submenu, you can choose for an trace list window showing only task switches (if any) or task switches together with default display.

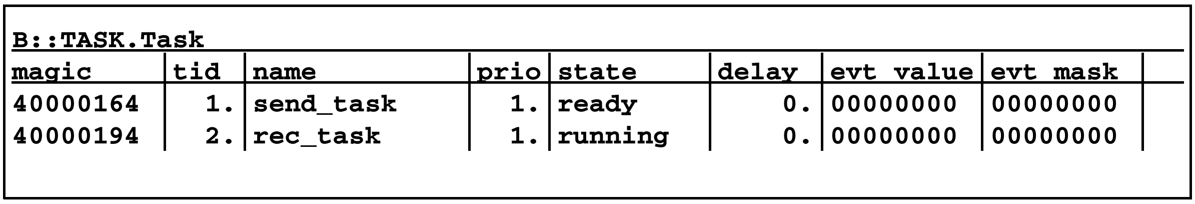
The 'Perf' menu contains additional submenus for task runtime statistics and statistics on task states.

TASK.Task

Display tasks

Format: **TASK.Task**

Displays the task table of RTX-ARM.



```
B::TASK.Task
magic    |tid|name          |prio|state     |delay|evt value|evt mask |
40000164 | 1.|send_task     |  1.|ready     |   0.|00000000 |00000000 |
40000194 | 2.|rec_task      |  1.|running   |   0.|00000000 |00000000 |
```

“magic” is an unique id, used by the RTOS Debugger to identify a specific task (address of the TCB).

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

RTX-ARM PRACTICE Functions

There are special definitions for RTX-ARM specific PRACTICE functions.

TASK.CONFIG(*<item>*)	Reports configuration parameters.
TASK.CONFIG(magic)	Returns the address for the magic number
TASK.CONFIG(magicsize)	Returns the size of the magic number (1,2 or 4)

Frequently-Asked Questions

No information available