LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
Rubus OS

MANUAL

OS Awareness Manual Rubus OS

OS Awareness Manual Rubus OS

Version 06-Jun-2024

History

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

Overview

The OS Awareness for Rubus OS (Arcticus Systems AB) 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 the Rubus OS is supported for the C167 microcontroller on the following versions:

- V1.0 (V1.1.0 beta) with large memory model
- V1.1 with small memory model

Manual Configuration

Format:	**TASK.CONFIG rubus** *<magic_address> <dpps> <args>*

<magic_address> The overall magic location ("*<magic_address>*") is currently not used. Specify "0".

<dpps> The *<dpps>* argument configures the data page settings of the application. Specify a long word, which least significant byte is the dpp0 content and which most significant byte is the dpp3 content. E.g. "03060500" means dpp0=0, dpp1=5, dpp2=6 and dpp3=3. If you don't know the dpp settings of your application, just start it for a while and check in the 'register' window the dpp's. If the parameter is "0", a linear DPP setup is assumed (i.e. "03020100").

<args> The additional arguments specify the symbols of the object lists. Use them as shown below:

```
bsVar redKernelVar redScheduleAttrList
blueKernelVar blueThreadAttrList blueMsgAttrList
blueMutexAttrList ioDevAttrList ioAttr
```

This command configures the OS Awareness for Rubus OS with manual setup.

The **TASK.CONFIG** command loads an extension definition file called "rubus.t32" (directory "~~/demo/c166/kernel/rubus/"). It contains all necessary extensions.

The configuration requires additional arguments. Specify them as shown above. They are pointers to internal tables of system variables.

```
; manual configuration for Rubus OS support
TASK.CONFIG rubus 0 03020100 bsVar redKernelVar redScheduleAttrList
blueKernelVar blueThreadAttrList blueMsgAttrList blueMutexAttrList
ioDevAttrList ioAttr
```

If you want to have dual port access for the display functions (display "On The Fly"), you have to map emulation memory to the address space of all used system tables.

See also the example "~~/demo/c166/kernel/rubus/rubus.cmm"

Format:	**TASK.CONFIG rubus 0** *<dpps>*
<dpps>:	*<dpp_settings>*

This command configures the OS Awareness for Rubus OS with automatic setup.

The **TASK.CONFIG** command loads an extension definition file called “rubus.t32” (directory “`~~/demo/c166/kernel/rubus`”). It contains all necessary extensions.

This configuration tries to locate the Rubus internals automatically. For this purpose the symbols mentioned in “**Manual Configuration**” must be loaded and accessible at any time, the OS Awareness is used.

Each **TASK.CONFIG** argument can be substituted by ’0’, which means that this argument will be searched and configured automatically.

If the application uses a linear DPP setting, you can omit all parameters for a fully automatic configuration:

```
; fully automatic configuration for Rubus support, linear DPPs
task.config rubus
```

If the application uses non-linear DPP settings, they must be specified by hand. See “**Manual Configuration**” for details on how to specify the DPPs.

```
; fully automatic configuration for Rubus support, non-linear DPPs
task.config rubus 0 03050600
```

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.

If you want to have dual port access for the display functions (display “On The Fly”), you have to map emulation memory to the address space of all used system tables.

See also the example “`~~/demo/c166/kernel/rubus/rubus.cmm`”

Hooks in Rubus OS

The variable “`bsVar`” is used to detect, whether blue or red kernel is running.
For detecting the running blue thread, the variable “`blueKernelVar`” is used.
For detecting the running red thread, either “redKernelVar” or “redStackFrame” is used.

Features

The OS Awareness for Rubus OS 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.MonDev	I/O devices
TASK.MonFile	Open files
TASK.MonLabel	Rubus Label
TASK.MonMsg	Blue message queues
TASK.MonMuteX	Blue mutexes
TASK.MonRSList	Red schedules
TASK.MonRSched	Red schedule threads
TASK.MonThread	Blue threads

For a description of the commands, refer to chapter “Rubus PRACTICE Commands”.

When working with emulation memory or shadow memory, these resources can be displayed “On The Fly”, i.e. while the target application is running, without any intrusion to the application. If using this dual port memory feature, be sure that emulation memory is mapped to all places, where Rubus holds its tables.

When working only with target memory, 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 Runtime Statistics

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

Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

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

Trace.List List.TASK DEFault	Display trace buffer and task switches
Trace.STATistic.TASK	Display task runtime statistic evaluation
Trace.Chart.TASK	Display task runtime timechart
Trace.PROfileSTATistic.TASK	Display task runtime within fixed time intervals statistically
Trace.PROfileChart.TASK	Display task runtime within fixed time intervals as colored graph
Trace.FindAll Address TASK.CONFIG(magic)	Display all data access records to the "magic" location
Trace.FindAll CYcle owner OR CYcle context	Display all context ID records

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

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches 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)".

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)".

Rubus specific Menu

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

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

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

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

- The **Trace** menu contains two new submenus:
  - **Rubus Selective** allows to program the analyzer to record only blue thread switches, red thread calls or both.
  - **Rubus List** displays the analyzer content with Rubus specific information.
- The **Perf** menu contains the additional submenus for thread runtime statistics, thread related function runtime statistics and statistics on thread states. For the function runtime statistics, prepare command files called “men_ptfp.cmm”, “men_pb.cmm” and “men_pr.cmm” are used. These command files must be adapted to your application.

TASK.MonDev I/O device list

Format: **TASK.MonDev**

Displays a table with all configured I/O devices.

'devInit' shows the address of the initialization routine and is mouse sensitive. I.e. the status line will show the symbolic information.

TASK.MonFile Open file list

Format: **TASK.MonFile**

Displays a table with all open I/O files.

TASK.MonLabel Rubus information

Format: **TASK.MonLabel**

Displays Rubus compilation information.

Format: **TASK.MonMsg** *<message>*

Displays information about message queues.

Without any parameter you get a table with all configured message queues.

Double clicking on a message queue name or ID gives you detailed information about a specific queue.

You can specify a specific queue as argument. The command accepts IDs (task.mm 41004) or names (task.mm "shellReply"). When specifying a name, be sure to use straight quotation marks, otherwise you will get wrong results.

Double clicking on the 'ptr' address opens a dump window with the buffer contents.

TASK.MonMuteX Blue MUTEX table

Format: **TASK.MonMuteX**

Displays a table with all configured mutexes.

TASK.MonRSched — Red thread table

Format: **TASK.MonRSched** *<schedule>*

Displays all configured red threads in a given red schedule.

You can specify a specific schedule by its name. When omitting the parameter, the current active schedule is displayed.

TASK.MonRSList — Red schedule table

Format: **TASK.MonRSList**

Displays a table with all configured red schedules.

TASK.MonThread — Blue thread table

Format: **TASK.MonThread**

Displays a table with all configured blue threads in Rubus.

The display is similar to the monitor service 'monThreadList'.

The states of the threads are displayed grey, if the application is running in real time, or if the application is halted, while the red kernel is running.

There are special definitions for Rubus specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

Syntax: **TASK.CONFIG(***<keyword>***)**

<keyword>: **bluemagic** | **bsmagic** | **magic** | **magicsize** | **redmagic**

Parameter and Description:

bluemagic	**Parameter Type**: String (***without*** quotation marks). Returns the address of the blue magic number.
bsmagic	**Parameter Type**: String (***without*** quotation marks). Returns the address of the basic magic number.
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).
redmagic	**Parameter Type**: String (***without*** quotation marks). Returns the address of the red magic number.

Return Value Type: Hex value.