OS Awareness Manual DSP/BIOS

OS Awareness Manual DSP/BIOS

OS Awareness Manual DSP/BIOS

Overview

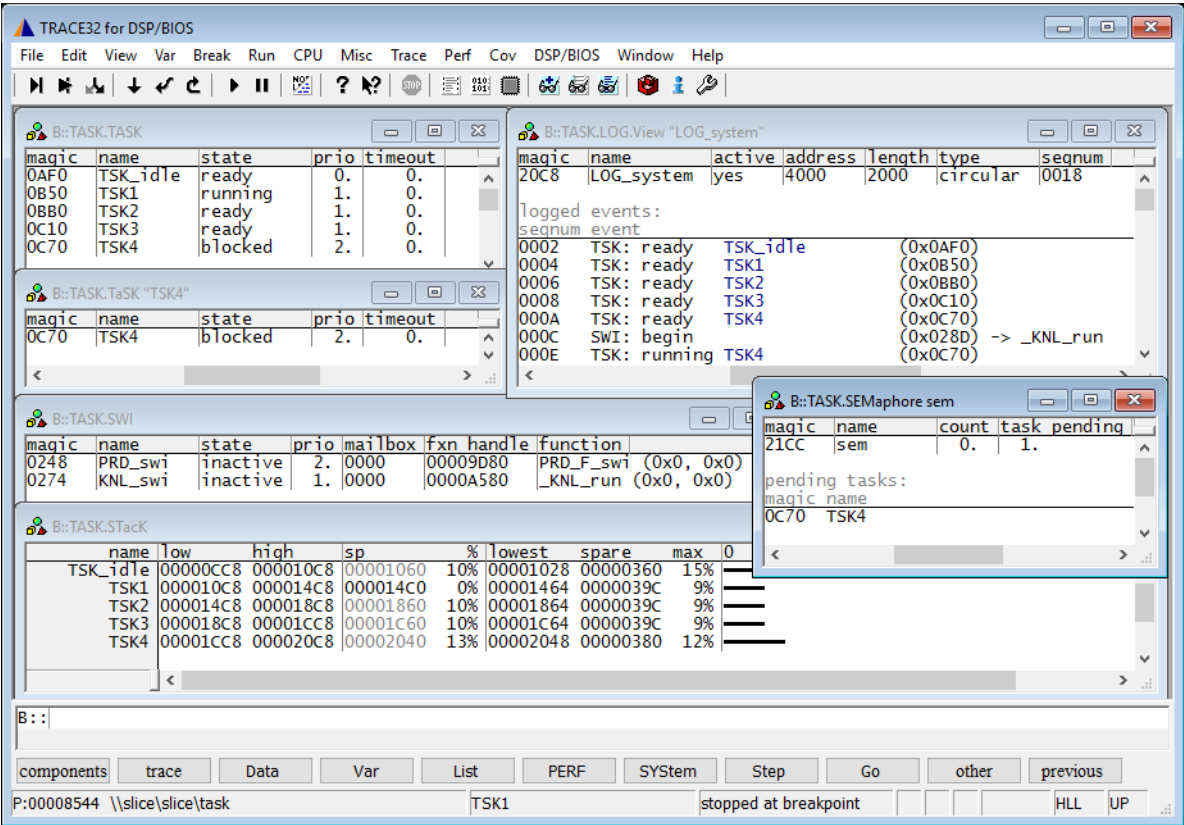The OS Awareness for DSP/BIOS contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands.

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 DSP/BIOS is supported for the following versions:

- Code Composer Studio v2 on TMS320C55xx and TMS320C64xx DSP.

Configuration

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

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

For system resource display, 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 bios**

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

See also the example "~~/demo/*<processor>*/kernel/bios/bios.cmm".

Quick Configuration Guide

To get a quick access to the features of the OS Awareness for DSP/BIOS with your application, follow the following roadmap:

1. Copy the files "`bios.t32`" and "`bios.men`" to your project directory
   (from TRACE32 directory "~~/demo/*<processor>*/kernel/bios").

2. Start the TRACE32 Debugger.

3. Load your application as normal.

4. Execute the command "`TASK.CONFIG bios`"

   (See "**Configuration**").

5. Execute the command "`MENU.ReProgram bios`"

   (See "**DSP/BIOS specific Menu**").

6. Start your application.

Now you can access the DSP/BIOS extensions through the menu.

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

Hooks & Internals in DSP/BIOS

No hooks are used in the kernel.

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

For retrieving the kernel data 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.

Used symbols:
`KNL_curtask, OBJ_table, KNL_swi, MEM_memtab`

Features

The OS Awareness for DSP/BIOS supports the following features.

Display of Kernel Resources

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

TASK.KerNeL	Kernel information
TASK.TaSK	Tasks
TASK.MailBoX	Mailboxes
TASK.SEMaphore	Semaphores
TASK.MEMory	Memory segments
TASK.SWI	SWIs

For a description of the commands, refer to chapter "**DSP/BIOS Commands**".

If your target CPU provides memory access while running (**SYStem.MemAccess Enable**), these resources can be displayed "On The Fly", i.e. while the target application is running, without any intrusion to the application.

If your target doesn't support this memory access, 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.

```
B::TASK.STacK
        name  low      high     sp       %  lowest   spare     max  0   10   20
TSK_idle 00000CC8 000010C8 00001060 10% 00001064 0000039C   9%
    TSK1 000010C8 000014C8 00001460 10% 00001464 0000039C   9%
    TSK2 000014C8 000018C8 00001860 10% 00001864 0000039C   9%
    TSK3 000018C8 00001CC8 00001C60 10% 00001C64 0000039C   9%
    TSK4 00001CC8 000020C8 00002060 10% 00002064 0000039C   9%
 (other)                   00002D00
```

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.

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.
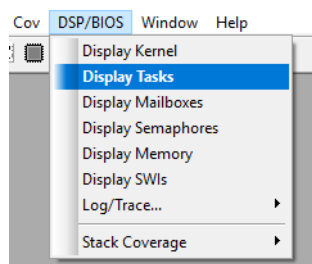
If **PERF** collects the PC for function profiling of processes in MMU-based operating systems (**SYStem.Option.MMUSPACES ON**), then you need to set **PERF.MMUSPACES**, too.

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

DSP/BIOS specific Menu

The menu file "bios.men" contains a menu with DSP/BIOS specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **DSP/BIOS**.

- The **Display** menu items launch the kernel resource display windows.

- The **Log/Trace** submenu allows to view the DSP/BIOS log and to enable/disable individual events.

- The **Stack Coverage** submenu starts and resets the DSP/BIOS specific stack coverage, and provide an easy way to add or remove tasks from the stack coverage window.
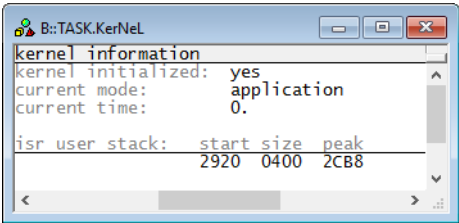
DSP/BIOS Commands

TASK.KerNeL — Display kernel information

Format:	**TASK.KerNeL**

Displays internal information about the current state of the kernel.
The display is similar to the "KNL" tab of the CCS Debugger.

TASK.LOG.DISable — Disable system log events

Format:	**TASK.LOG.DISable** [**all**	**SWIlog**	**PRDlog**	**CLKlog**	**TSKlog**	**SWIAcc**	**PRDAcc**	**PIPAcc**	**HWIAcc**	**TSKAcc**	**User0**	**User1**	**User2**]

Disables tracing of the specified event in the DSP/BIOS system log.

TASK.LOG.ENable — Enable system log events

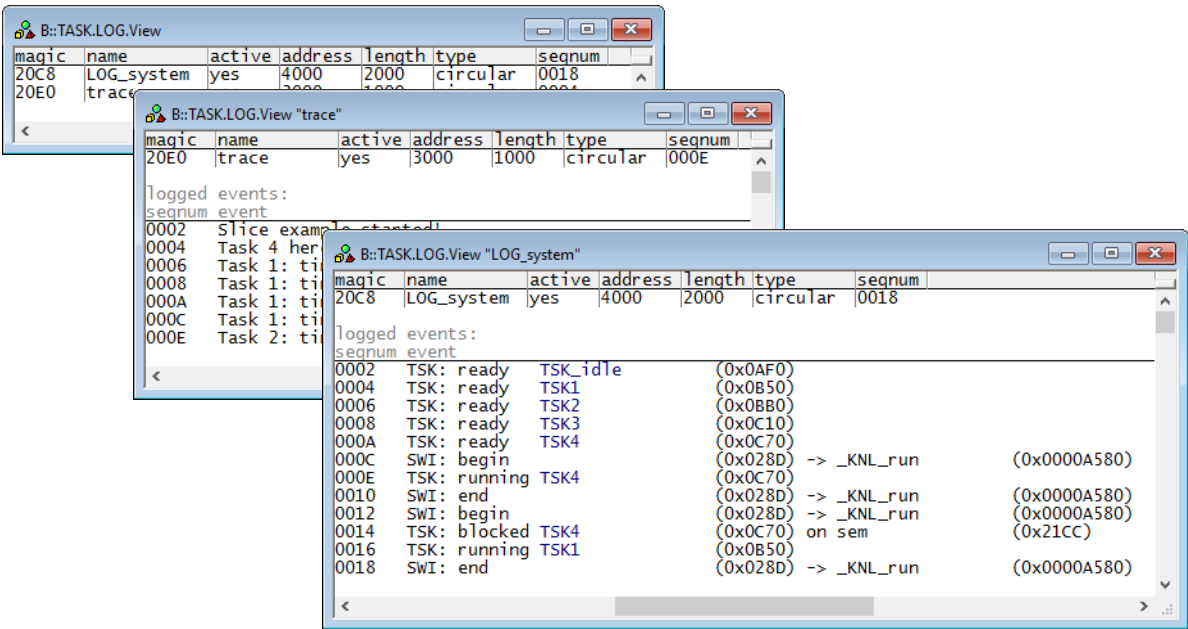| Format: | **TASK.LOG.ENable** [**all** | **SWIlog** | **PRDlog** | **CLKlog** | **TSKlog** | **SWIAcc** | **PRDAcc** | **PIPAcc** | **HWIAcc** | **TSKAcc** | **User0** | **User1** | **User2**] |
|---|---|

Enables tracing of the specified event in the DSP/BIOS system log.

Format:	**TASK.LOG.View** [*<log>*]

Displays a table with all created Logs of DSP/BIOS.

The display is similar to the "SWI" tab of the CCS Debugger.

Without any arguments, a table with all created logs will be shown.
Specify a log name or magic number to display the content of this log.

"magic" (= handle) is a unique ID, used by the OS Awareness to identify a specific log (address of the log object).
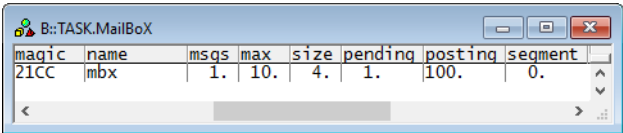
The field "magic" is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

TASK.MailBoX

Format:	**TASK.MailBoX** [*<mailbox>*]

Displays the mailbox table of DSP/BIOS or detailed information about one specific mailbox.

The display is similar to the "MBX" tab of the CCS Debugger.

Without any arguments, a table with all created mailboxes will be shown.
Specify a mailbox name or magic number to display detailed information on that mailbox.
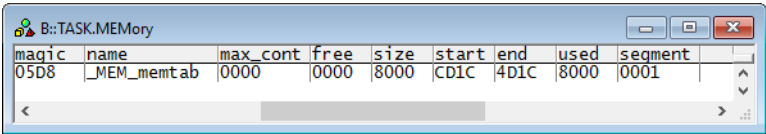

"magic" is a unique ID, used by the OS Awareness to identify a specific thread (address of the mailbox object).

The fields "magic" and "name" are mouse sensitive. Double-clicking on them opens appropriate windows.

TASK.MEMory

Format:	**TASK.MEMory**

Displays a table with all created memory segments of DSP/BIOS.

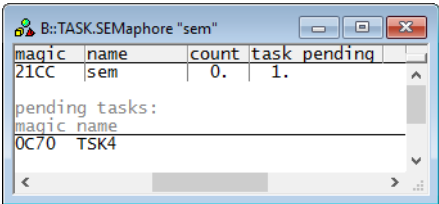The display is similar to the "SEM" tab of the CCS Debugger.

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

Displays the semaphore table of DSP/BIOS or detailed information about one specific semaphore.

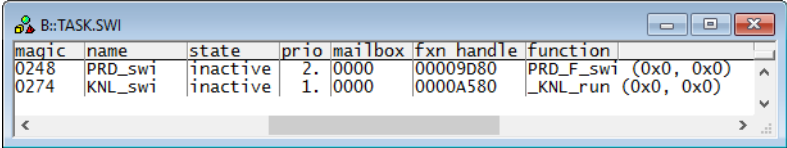The display is similar to the "SEM" tab of the CCS Debugger.

Without any arguments, a table with all created semaphores will be shown.
Specify a semaphore name or magic number to display detailed information on that port.

```
B::TASK.SEMaphore "sem"
magic   name      count  task pending
21CC    sem         0.      1.

pending tasks:
magic name
0C70   TSK4
```

Format:	**TASK.SWI**

Displays a table with all created SWIs of DSP/BIOS.

The display is similar to the "SWI" tab of the CCS Debugger.

```
B::TASK.SWI
magic  name      state     prio  mailbox  fxn handle  function
0248   PRD_swi   inactive   2.   0000     00009D80    PRD_F_swi (0x0, 0x0)
0274   KNL_swi   inactive   1.   0000     0000A580    _KNL_run (0x0, 0x0)
```
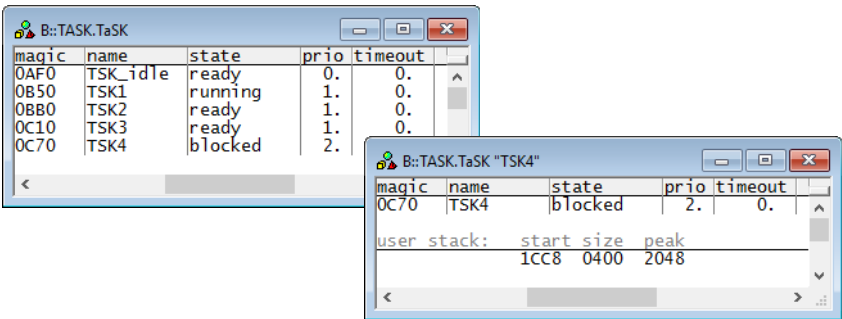
| Format: | **TASK.TaSK** [*<task>*] |

Displays the task table of DSP/BIOS or detailed information about one specific task.

The display is similar to the "TSK" tab of the CCS Debugger.

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" (= handle) is a unique ID, used by the OS Awareness to identify a specific task (address of the task object).

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

DSP/BIOS PRACTICE Functions

Currently, there are no special definitions for DSP/BIOS specific PRACTICE functions.

See also **general TASK functions**.