OS Awareness Manual AMX

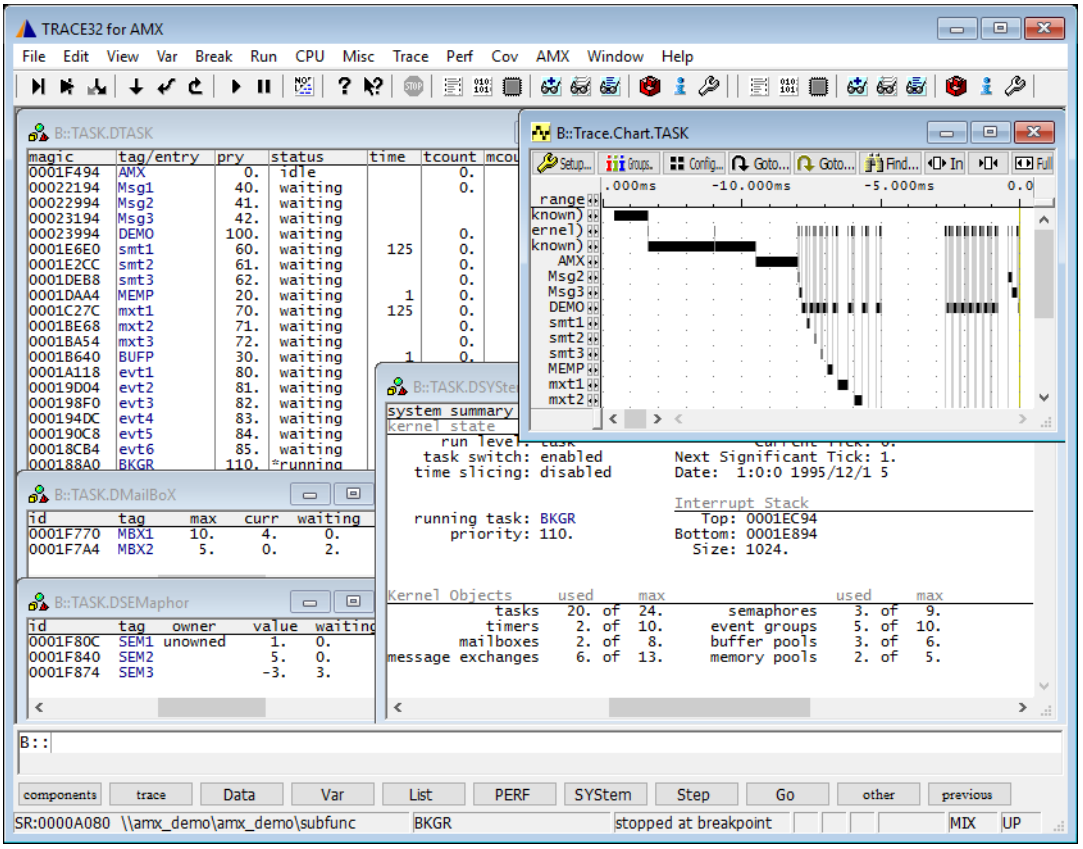
OS Awareness Manual AMX

Version 18-Nov-2020

History

28-Aug-18 The title of the manual was changed from “RTOS Debugger for *<x>*” to “OS Awareness Manual *<x>*”.

Overview



The OS Awareness for AMX 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**.

- **"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 AMX is supported for:

- AMX Version 3.04a on the Freescale Semiconductor 68332
- AMX on the ARM 7
- AMX on Freescale Semiconductor PowerPC

Configuration

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

Automatic configuration tries to locate the AMX internals automatically. For this purpose all symbol tables must 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 must 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). In case of a ICE or FIRE, you have to map emulation or shadow memory to the address space of all used system tables.

Manual Configuration

Format:	**TASK.CONFIG amx.t32** <*magic_address*> <*args*>

<*magic_address*>	Specifies a memory location that contains the current running task. This address can be found at “`cj_kdata+14`”.
<*args*>	The configuration requires one additional argument that specifies an AMX internal pointer. Give the label “`cj_kdatp`”.

Manual configuration for the OS Awareness for AMX can be used to explicitly define some memory locations. It is recommended to use automatic configuration.

```
; manual configuration for AMX support
task.config amx.t32 cj_kdata+14 cj_kdatp
```

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

Automatic Configuration

Format:	**TASK.CONFIG amx.t32**

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:

```
; fully automatic configuration for AMX support
task.config amx
```

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 OS Awareness for AMX with your application, follow the following roadmap:

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

Now you can access the AMX extensions through the menu.

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

All kernel resources are accessed through the kernel data pointer “cj_kdatp”.
The magic location is calculated from “(*c_kdatp+0x14)”.
For detecting a message exchanger task, the entry point of that task is compared to the message exchanger task entry point “cj_kpmxtask”). If this symbol is not available, the message exchanger tasks won’t be detected.
In the statistics evaluations, the kernel state is derived from the location at “(*cj_kdatp)”.

Features

The OS Awareness for AMX supports the following features.

Display of Kernel Resources

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

TASK.DSYStem	system state
TASK.DTask	Tasks
TASK.DTIMer	Timer
TASK.DMailBoX	Mailboxes
TASK.DESChange	Message exchanges
TASK.DSEMaphor	Semaphores
TASK.DEVent	Event groups
TASK.DBPool	Buffer pools
TASK.DMPool	Memory pools

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

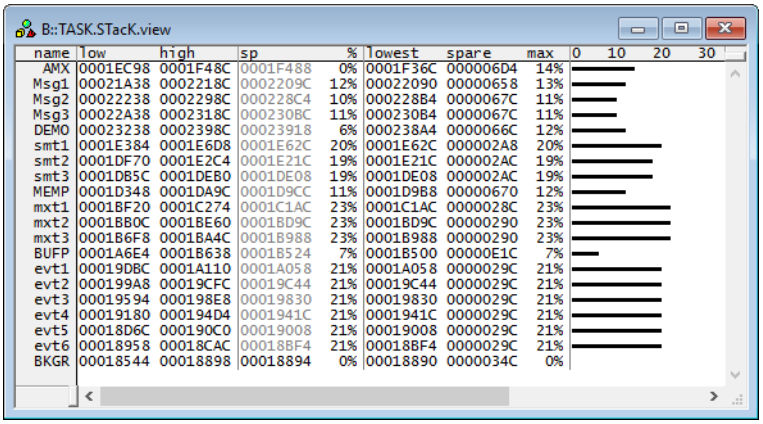
Task Stack Coverage

For stack usage coverage of the 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, 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** 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.

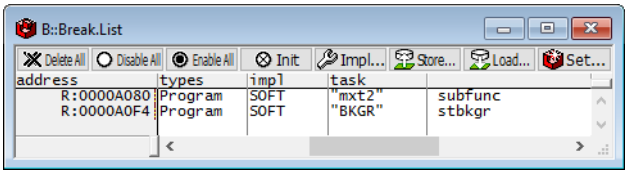
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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

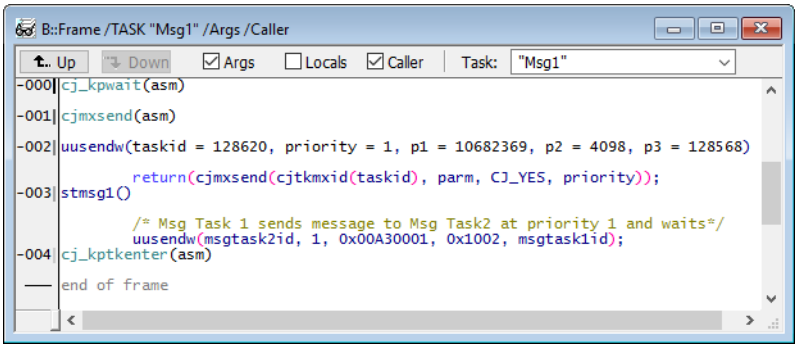
To display the call stack of a specific task, use the following command:

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.

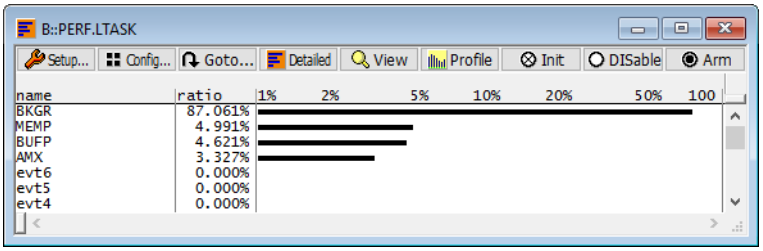


Dynamic Task Performance Measurement

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).



Task Runtime Statistics

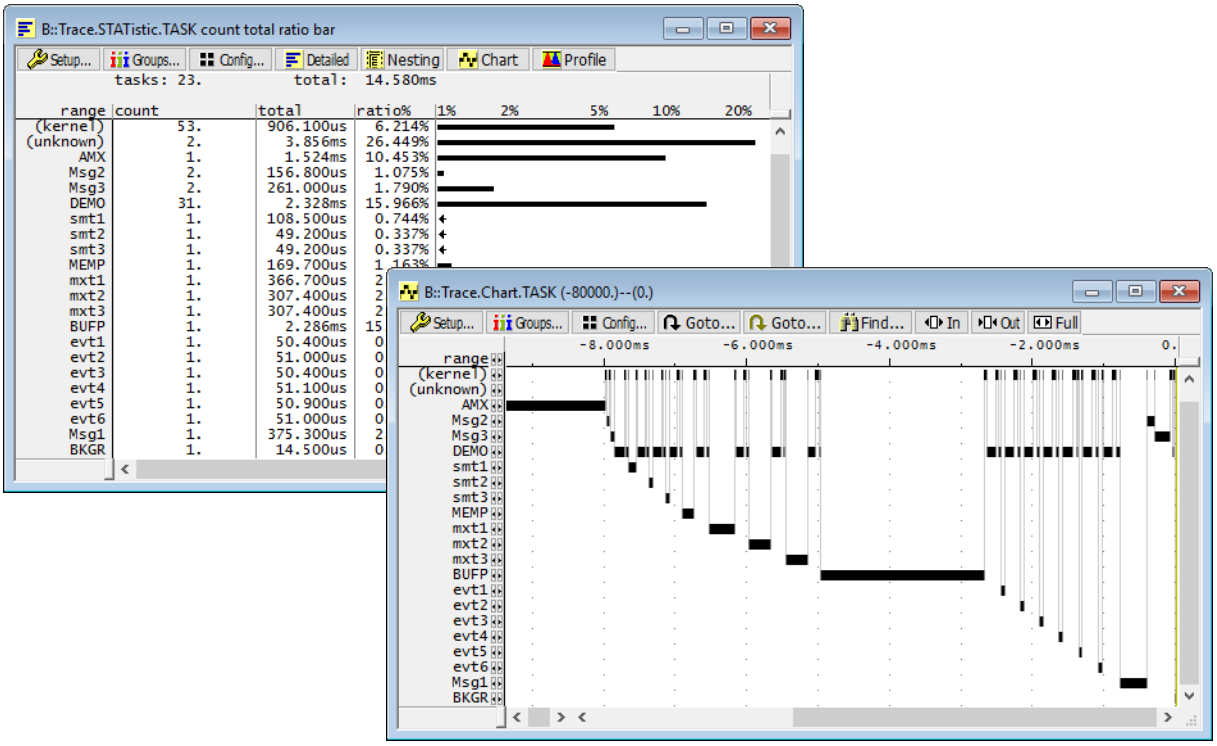
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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



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
```

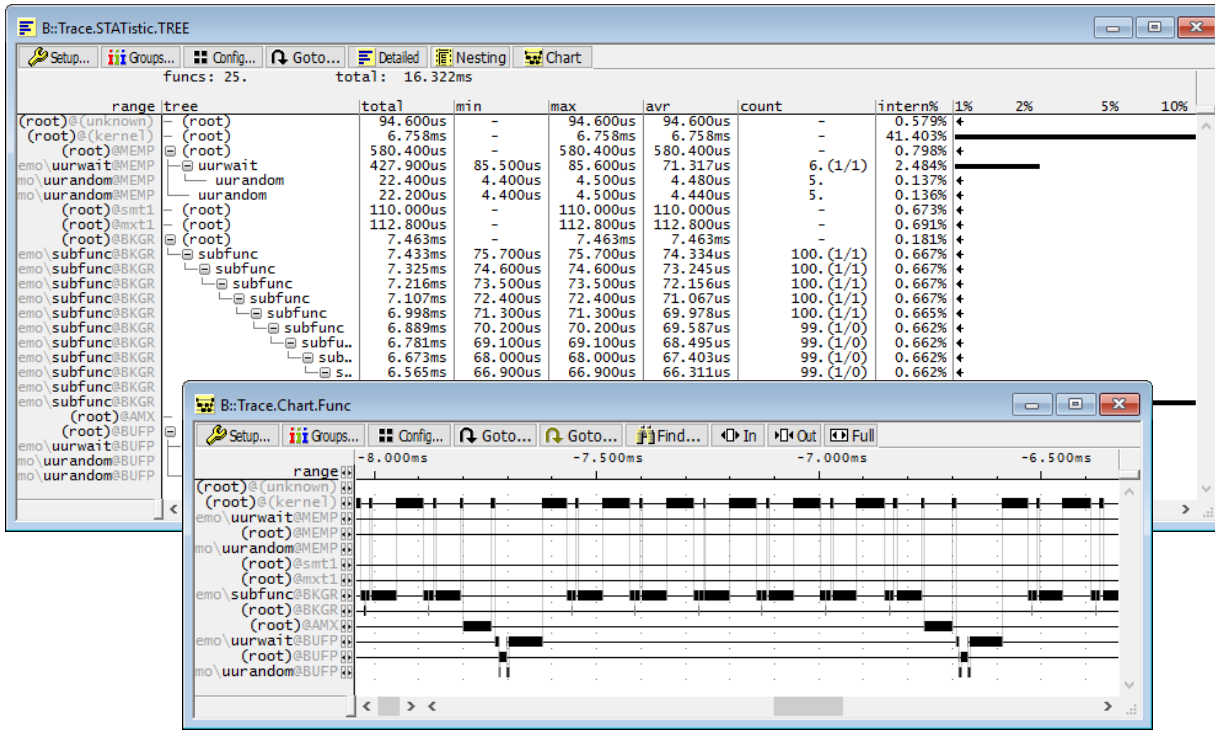
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

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

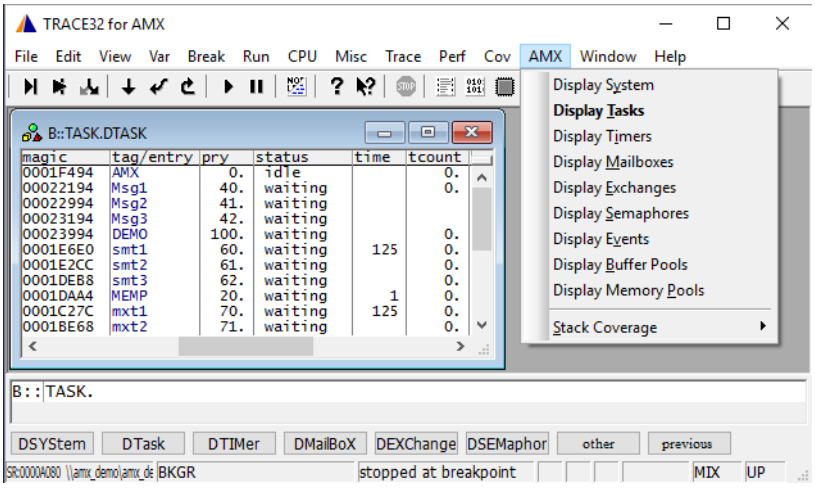
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat runtime timechart

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



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

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



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

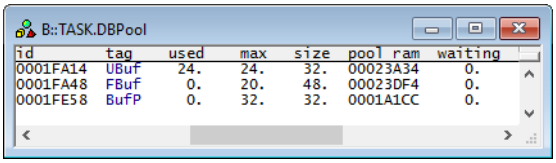
In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- The **Trace** -> **List** submenu is changed. You can additionally choose if you want a trace list window to show only task switches (if any) or task switches and defaults.
- The **Perf** menu contains the additional submenus for task runtime statistics, task-related function runtime statistics and statistics on task states. For the function runtime statistics, a PRACTICE script file called “men_ptfp.cmm” is used. This script file must be adapted to your application.

TASK.DBPool — Display buffer pools

Format: **TASK.DBPool**

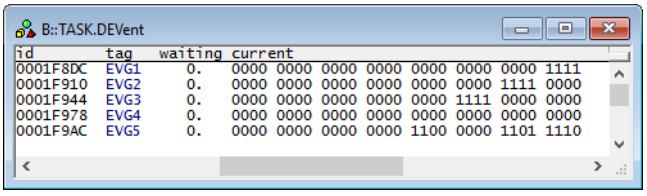
Displays a table with all created AMX buffer pools.



TASK.DEVent — Display event groups

Format: **TASK.DEVent**

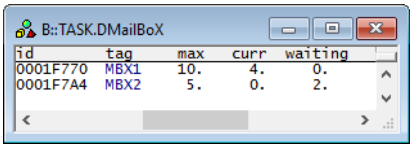
Displays a table with all created AMX event groups.



TASK.DEXChange — Display message exchanges

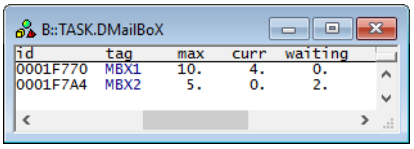
Format: **TASK.DEXChange**

Displays a table with all created AMX message exchanges.



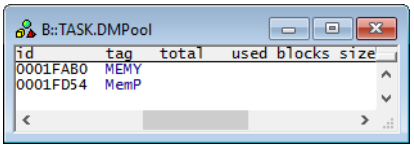
Format: **TASK.DMailBoX**

Displays a table with all created AMX mailboxes.



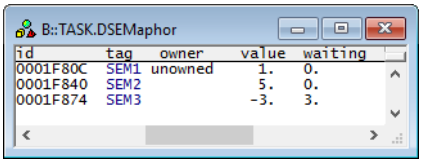
Format: **TASK.DMPool**

Displays a table with all created AMX memory pools.



Format: **TASK.DSEMaphore**

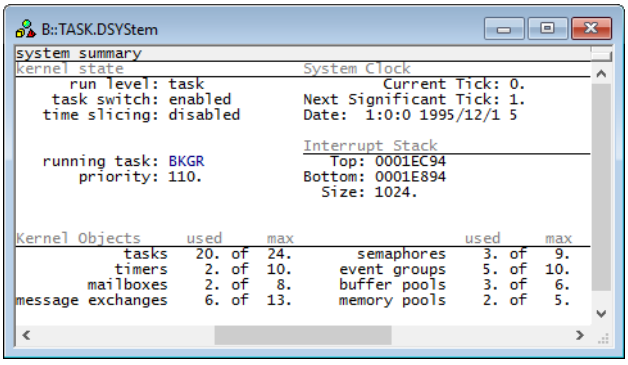
Displays a table with all created AMX semaphores.



TASK.DSYStem Display system state

Format: **TASK.DSYStem**

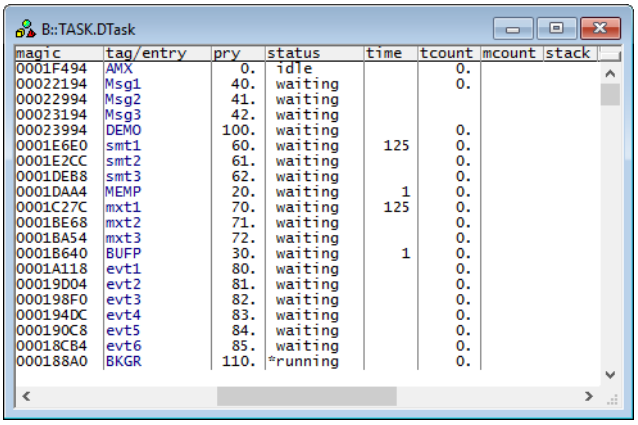
Displays a system state summary for the current AMX system state.



Format:	**TASK.DispTask** [*<task>*]
<task>:	*<task_magic>* | *<task_id>* | *<task_name>*

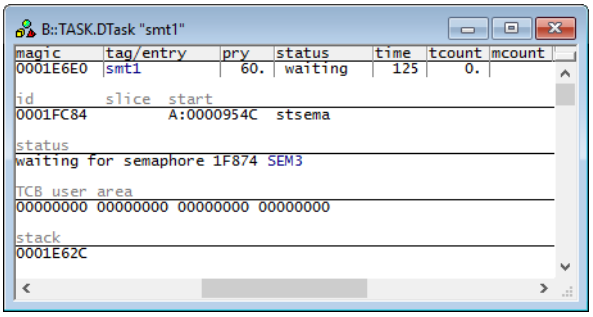
Displays a table with all AMX tasks or one task in detail.

Without any parameters, a summary table of all created tasks is shown.



The magic number is a unique ID to the OS Awareness to specify a specific task. It is **not** equal to the AMX task ID. A double click on the magic number or on the tag opens the detailed task window.

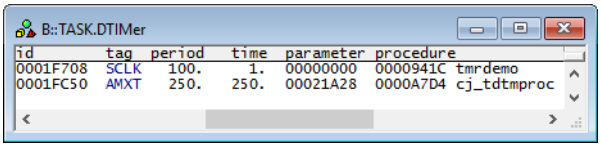
If you specify a task magic number, a task ID or a task tag as parameter, this task is shown in detailed. Enclose a task tag in quotation marks. If a numerical parameter is detected to be a AMX task ID, this one will be used. All other numerical parameters are supposed to be a task magic number and are not checked for validation.



Format: **TASK.DTIMer**

Displays a table showing all defined AMX timers.

Double click on the parameter to see a dump window on this address. Double click on the procedure to see a list window on this address.



TASK.TASKState Mark task state words

Format: **TASK.TASKState**

This command sets Alpha breakpoints on all task status words.

The statistic evaluation of task states (see Task State Analysis) requires recording of the accesses to the task state words. By setting Alpha breakpoints to these words and selectively recording Alpha's, you can do a selective recording of task state transitions.

Because setting the Alpha breakpoints by hand is very hard to do, this utility command automatically sets the Alpha's to the status words of all tasks currently created. It does NOT set breakpoints to tasks that terminated or haven't yet been created.

AMX PRACTICE Functions

There are special definitions for AMX specific PRACTICE functions.

See also **general TASK functions**.

TASK.CONFIG() OS Awareness configuration information

Syntax: **TASK.CONFIG(magic | magicsize | kdata)**

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).
kdata	**Parameter Type**: String (***without*** quotation marks). Returns the address of the kernel data area.

Return Value Type: Hex value.

Frequently-Asked Questions

No information available