OS Awareness Manual CMX

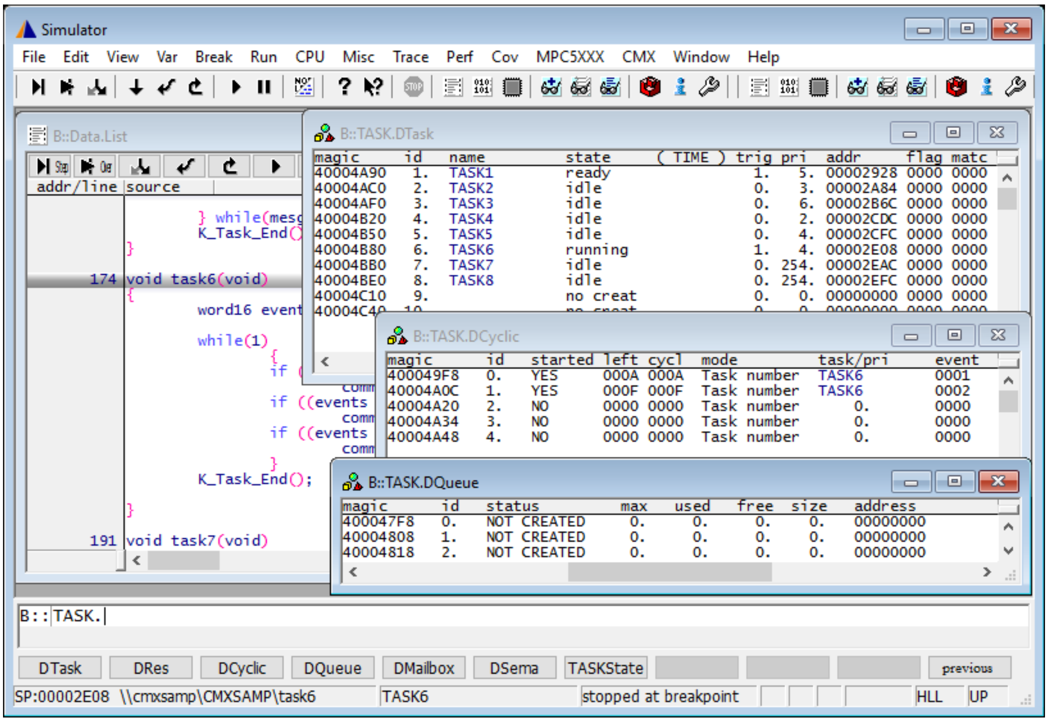
OS Awareness Manual CMX

OS Awareness Manual CMX

Version 06-Jun-2024

NOTE:	**This manual contains a TRACE32 specific CMX-RTX awareness that is outdated and no longer maintained. Instead, CMX supports the OSEK-ORTI standard to provide OS Awareness in debuggers. Please see http://www.cmx.com/cmxkaware.htm. In TRACE32, please use the ORTI awareness for CMX-RTX. See the document in the TRACE32 installation folder, pdf/rtos_orti.pdf.**

Overview


Simulator
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX CMX Window Help
B::Data.List
Step Over
B::TASK.DTask
B::TASK.DCyclic
B::TASK.DQueue
DTask DRes DCyclic DQueue DMailbox DSema TASKState previous
SP:00002E08 \\cmxsamp\CMXSAMP\task6 TASK6 stopped at breakpoint HLL UP

```
addr/line source
                } while(mesg
                K_Task_End()
           }
       174 void task6(void)
           {
                word16 event

                while(1)
                       {
                       if (
                               comm
                       if ((events
                               comm
                       if ((events
                               comm
                       }
                K_Task_End();
           }
       191 void task7(void)
```

```
magic      id   name        state    ( TIME ) trig pri  addr     flag matc
40004A90   1.   TASK1       ready               1.   5. 00002928 0000 0000
40004AC0   2.   TASK2       idle                0.   3. 00002A84 0000 0000
40004AF0   3.   TASK3       idle                0.   6. 00002B6C 0000 0000
40004B20   4.   TASK4       idle                0.   2. 00002CDC 0000 0000
40004B50   5.   TASK5       idle                0.   4. 00002CFC 0000 0000
40004B80   6.   TASK6       running             1.   4. 00002E08 0000 0000
40004BB0   7.   TASK7       idle                0. 254. 00002EAC 0000 0000
40004BE0   8.   TASK8       idle                0. 254. 00002EFC 0000 0000
40004C10   9.               no creat            0.   0. 00000000 0000 0000
```

```
magic     id   started left cycl  mode          task/pri      event
400049F8  0.   YES     000A 000A  Task number   TASK6         0001
40004A0C  1.   YES     000F 000F  Task number   TASK6         0002
40004A20  2.   NO      0000 0000  Task number         0.      0000
40004A34  3.   NO      0000 0000  Task number         0.      0000
40004A48  4.   NO      0000 0000  Task number         0.      0000
```

```
magic     id   status        max    used    free   size   address
400047F8  0.   NOT CREATED    0.      0.      0.     0.   00000000
40004808  1.   NOT CREATED    0.      0.      0.     0.   00000000
40004818  2.   NOT CREATED    0.      0.      0.     0.   00000000
```

```
B::TASK.
```

The OS Awareness for CMX 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 CMX is supported for the following versions:

- CMX-RTX 4.0 on Infineon 8051, C16x and Zilog Z180/64180
- CMX-RTX 5.3 on Infineon C16x, Renesas H8/300H, H8, SH and NEC V850
- CMX-RTXS 1.0 on Infineon C166/C167

Configuration

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

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

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.

For system resource display and analyzer 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. 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 cmxrtx**

Quick Configuration Guide

To access all features of the OS Awareness you should follow the following roadmap:

1. Run the demo script (~~/demo/*<processor>*/kernel/cmx_rtx/cmx.cmm). Start the demo with “`do cmx`” and “`go`”. The result should be a list of tasks, which continuously change their state.
2. Make a copy of the “cmx.cmm” PRACTICE script file. Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the kernel resources and use the analyzer functions.

Hooks & Internals in CMX

No hooks are used in the kernel.
For detecting the running task, the variable 'activetcb' is used.

Features

The OS Awareness for CMX supports the following features.

CMXBug Terminal Emulation

The terminal emulation window can be used to communicate with the target resident CMX debugger, called CMXBug. The communication via two memory cells requires no external interface. See the **TERM** command group for a description of the terminal emulation. On request we can provide you with the source code for the target interface routine.

Display of Kernel Resources

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

TASK.DCyclic	Cyclic timers
TASK.DMailbox	Mailboxes
TASK.DQueue	Queues
TASK.DRes	Resources
TASK.DSema	Semaphores
TASK.DTask	Tasks

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

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.

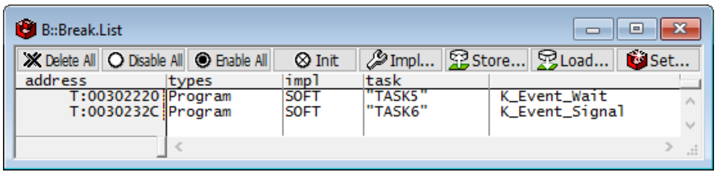
On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering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



```
address          types     impl    task
      T:00302220 Program   SOFT    "TASK5"        K_Event_Wait
      T:0030232C Program   SOFT    "TASK6"        K_Event_Signal
```

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

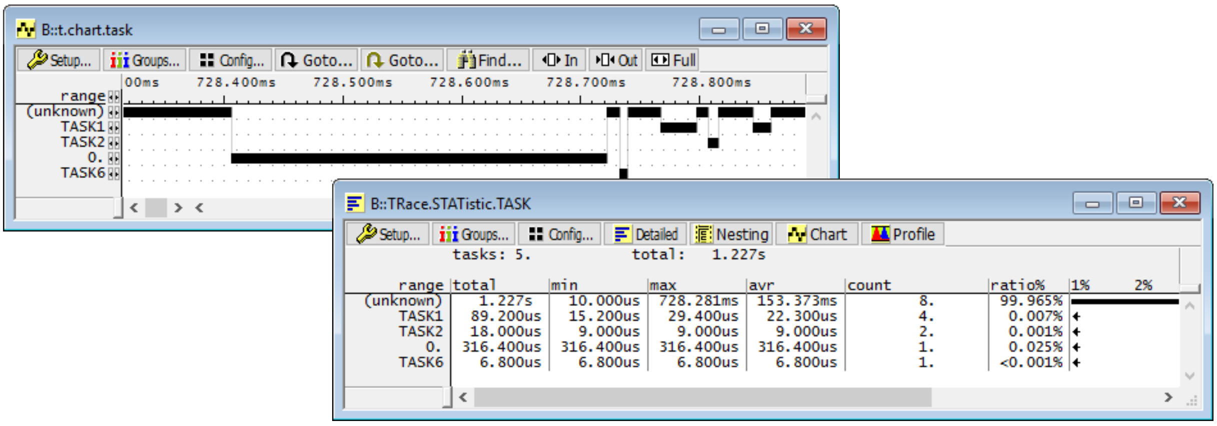
NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software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



```
      tasks: 5.           total:   1.227s

     range |total     |min       |max       |avr       |count          |ratio%   |1%     2%
 (unknown) |   1.227s |10.000us  |728.281ms |153.373ms |        8.     | 99.965% |
     TASK1 | 89.200us |15.200us  | 29.400us | 22.300us |        4.     |  0.007% |
     TASK2 | 18.000us | 9.000us  |  9.000us |  9.000us |        2.     |  0.001% |
        0. |316.400us |316.400us |316.400us |316.400us |        1.     |  0.025% |
     TASK6 |  6.800us | 6.800us  |  6.800us |  6.800us |        1.     | <0.001% |
```

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
```

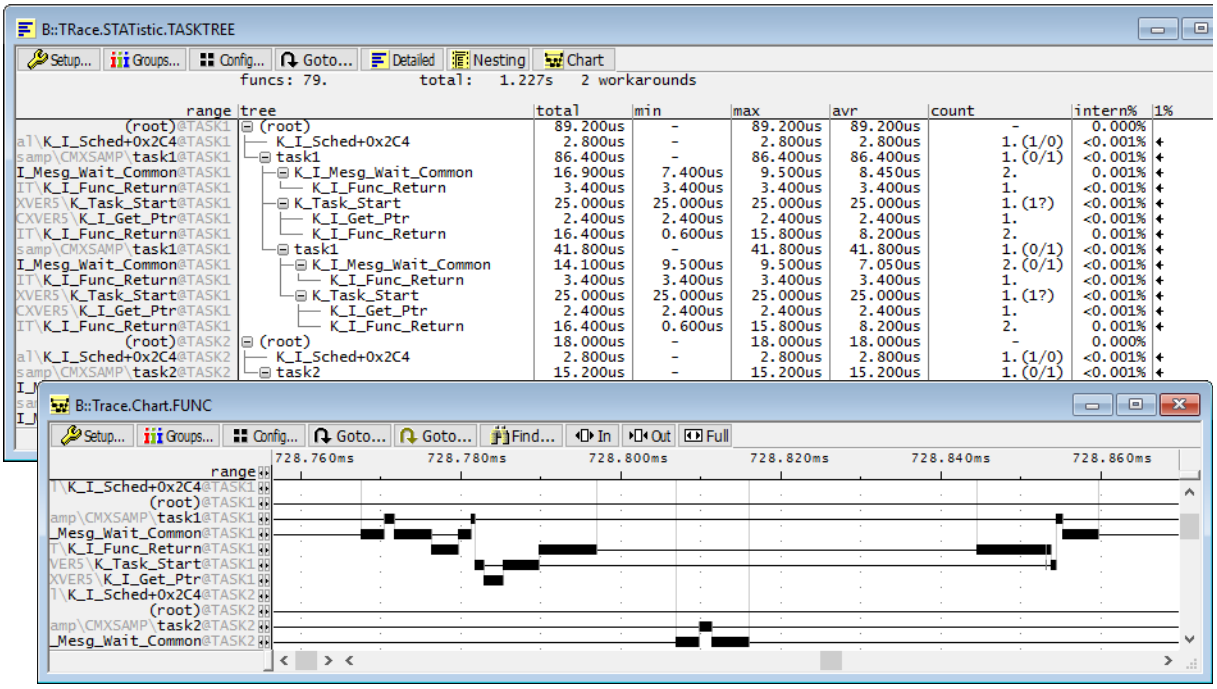
To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; 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)".



```
                    funcs: 79.        total:   1.227s   2 workarounds

                  range |tree                        |total     |min       |max       |avr       |count        |intern% |1%
          (root)@TASK1  ⊟ (root)                      89.200us       -     89.200us   89.200us            -      0.000%
al\K_I_Sched+0x2C4@TASK1 ├── K_I_Sched+0x2C4           2.800us       -      2.800us    2.800us      1.(1/0)   <0.001%
samp\CMXSAMP\task1@TASK1 └⊟ task1                     86.400us       -     86.400us   86.400us      1.(0/1)   <0.001%
I_Mesg_Wait_Common@TASK1    ├⊟ K_I_Mesg_Wait_Common   16.900us   7.400us    9.500us    8.450us      2.          0.001%
IT\K_I_Func_Return@TASK1    │ └── K_I_Func_Return      3.400us   3.400us    3.400us    3.400us      1.        <0.001%
XVER5\K_Task_Start@TASK1    ├⊟ K_Task_Start           25.000us  25.000us   25.000us   25.000us      1.(1?)    <0.001%
CXVER5\K_I_Get_Ptr@TASK1    │ ├── K_I_Get_Ptr          2.400us   2.400us    2.400us    2.400us      1.        <0.001%
IT\K_I_Func_Return@TASK1    │ └── K_I_Func_Return     16.400us   0.600us   15.800us    8.200us      2.          0.001%
samp\CMXSAMP\task1@TASK1    └⊟ task1                  41.800us       -     41.800us   41.800us      1.(0/1)   <0.001%
I_Mesg_Wait_Common@TASK1      ├⊟ K_I_Mesg_Wait_Common 14.100us   9.500us    9.500us    7.050us      2.(0/1)   <0.001%
IT\K_I_Func_Return@TASK1      │ └── K_I_Func_Return    3.400us   3.400us    3.400us    3.400us      1.        <0.001%
XVER5\K_Task_Start@TASK1      └⊟ K_Task_Start         25.000us  25.000us   25.000us   25.000us      1.(1?)    <0.001%
CXVER5\K_I_Get_Ptr@TASK1        ├── K_I_Get_Ptr        2.400us   2.400us    2.400us    2.400us      1.        <0.001%
IT\K_I_Func_Return@TASK1        └── K_I_Func_Return   16.400us   0.600us   15.800us    8.200us      2.          0.001%
          (root)@TASK2  ⊟ (root)                      18.000us       -     18.000us   18.000us            -      0.000%
al\K_I_Sched+0x2C4@TASK2 ├── K_I_Sched+0x2C4           2.800us       -      2.800us    2.800us      1.(1/0)   <0.001%
samp\CMXSAMP\task2@TASK2 └⊟ task2                     15.200us       -     15.200us   15.200us      1.(0/1)   <0.001%
```

```
                          range
1\K_I_Sched+0x2C4@TASK1
              (root)@TASK1
amp\CMXSAMP\task1@TASK1
_Mesg_Wait_Common@TASK1
T\K_I_Func_Return@TASK1
VER5\K_Task_Start@TASK1
XVER5\K_I_Get_Ptr@TASK1
1\K_I_Sched+0x2C4@TASK2
              (root)@TASK2
amp\CMXSAMP\task2@TASK2
_Mesg_Wait_Common@TASK2
```

CMX specific Menu

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

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

- The **CMXBug Terminal** menu item (if available) brings up a terminal emulation window, which communicates with the preconfigured CMXBug debugger.
- **Break to CMXBug** activates CMXBug.
- The **Display** menu item launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the CMX 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 and defaults.
- The **Perf** menu contains the additional submenus for task runtime statistics, task-related function runtime statistics and statistics on task states.

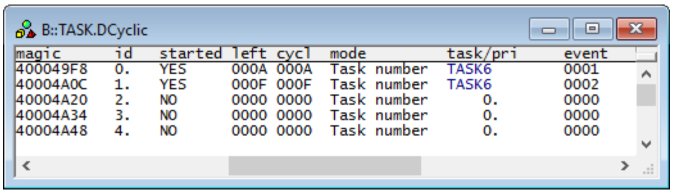
TASK.DCyclic

Display cyclic timers

Format: **TASK.DCyclic**

Displays information about all cyclic timers.

The display is similar to the CMXBug 'CYCLIC TIMER' dump.



```
magic     id    started left cycl  mode        task/pri    event
400049F8  0.    YES     000A 000A  Task number TASK6       0001
40004A0C  1.    YES     000F 000F  Task number TASK6       0002
40004A20  2.    NO      0000 0000  Task number       0.    0000
40004A34  3.    NO      0000 0000  Task number       0.    0000
40004A48  4.    NO      0000 0000  Task number       0.    0000
```

For an explanation of the items, see the CMXBug manual, CYCLIC TIMERS function.

A corresponding task is displayed with its name, if available. Otherwise the task ID is displayed.

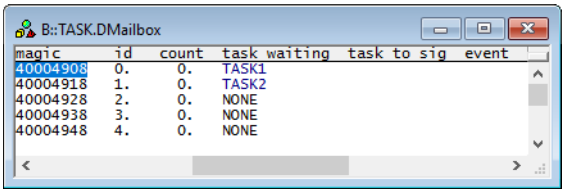
TASK.DMailbox

Display mailboxes

Format: **TASK.DMailbox** *<mailbox>*

Displays a table with all CMX mailboxes or one specific mailbox in detail.

The display of this table is similar to the CMXBug 'MAILBOX' dump



```
magic     id    count  task waiting  task to sig  event
40004908  0.       0.  TASK1
40004918  1.       0.  TASK2
40004928  2.       0.  NONE
40004938  3.       0.  NONE
40004948  4.       0.  NONE
```

For an explanation of the items, see the CMXBug manual, MAILBOXES function.

The task are displayed with their names, if available. Otherwise the task IDs are displayed.

The 'magic' field is mouse sensitive, a double click on it will give you detailed information about this specific mailbox.

For a detailed information you can also specify a magic of a mailbox as argument to this function. You will get a list with all messages in this mailbox.

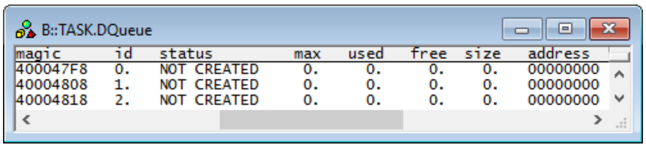
The message address is mouse sensitive. Double clicking on it shows a memory dump of this address.

TASK.DQueue Display queues

Format: **TASK.DQueue**

Displays a table with all CMX queues.

The display is similar to the CMXBug 'QUEUE' dump.



```
magic     id   status        max    used   free   size   address
400047F8  0.   NOT CREATED   0.     0.     0.     0.     00000000
40004808  1.   NOT CREATED   0.     0.     0.     0.     00000000
40004818  2.   NOT CREATED   0.     0.     0.     0.     00000000
```

For an explanation of the items, see the CMXBug manual, QUEUE function.

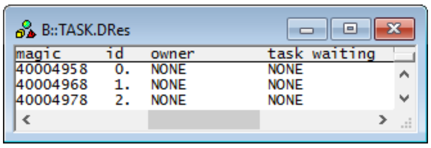
The 'address' field is mouse sensitive. Double clicking on it shows you a 'dump' window of this address.

TASK.DRes Display resources

Format: **TASK.DRes**

Displays all resources specified in CMX.

The display is similar to the CMXBug 'RESOURCE' dump



```
magic     id   owner      task waiting
40004958  0.   NONE       NONE
40004968  1.   NONE       NONE
40004978  2.   NONE       NONE
```

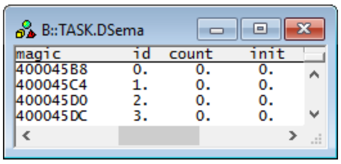
The task are displayed with their names, if available. Otherwise the task IDs are displayed.

A maximum number of five waiting tasks will be shown. An ending 'more…' will indicate that there are more than five waiting tasks.

TASK.DSema Display semaphores

Format: **TASK.DSema**

Displays a table with all CMX semaphores.



```
B::TASK.DSema
magic      id   count    init
400045B8   0.     0.      0.
400045C4   1.     0.      0.
400045D0   2.     0.      0.
400045DC   3.     0.      0.
```

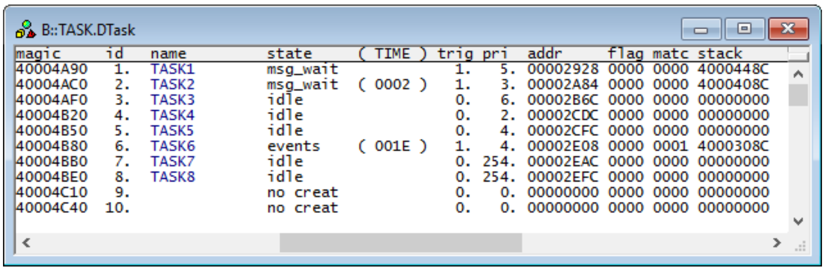
For an explanation of the items, see the CMXBug manual, SEMAPHORE function.

TASK.DTask Display tasks

Format: **TASK.DTask**

Displays the task table of MQX.

The display is similar to the CMXBug 'TASK' dump.



```
B::TASK.DTask
magic     id   name        state     ( TIME ) trig pri  addr     flag matc stack
40004A90   1.  TASK1       msg_wait             1.   5. 00002928 0000 0000 4000448C
40004AC0   2.  TASK2       msg_wait  ( 0002 )   1.   3. 00002A84 0000 0000 4000408C
40004AF0   3.  TASK3       idle                 0.   6. 00002B6C 0000 0000 00000000
40004B20   4.  TASK4       idle                 0.   2. 00002CDC 0000 0000 00000000
40004B50   5.  TASK5       idle                 0.   4. 00002CFC 0000 0000 00000000
40004B80   6.  TASK6       events    ( 001E )   1.   4. 00002E08 0000 0001 4000308C
40004BB0   7.  TASK7       idle                 0. 254. 00002EAC 0000 0000 00000000
40004BE0   8.  TASK8       idle                 0. 254. 00002EFC 0000 0000 00000000
40004C10   9.              no creat             0.   0. 00000000 0000 0000 00000000
40004C40  10.              no creat             0.   0. 00000000 0000 0000 00000000
```

The magic specifies the value, with which the OS Awareness identifies a specific task.
The task name is only displayed, if the task was named with the 'cxtname' function.
The stack column shows the stack bottom (high) address; this field is mouse sensitive.
For an explanation of the other items, see the CMXBug manual, TASKS function.

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

CMX PRACTICE Functions

There are special definitions for CMX specific PRACTICE functions.

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.

TASK.STACK() Stack information of a task

Syntax: **TASK.STACK(bottom | pointer,** *<task_magic>***)**

Parameter and Description:

<task_magic>	**Parameter Type**: Decimal or hex or binary value. Magic number of the task to query the information.
bottom	**Parameter Type**: String (***without*** quotation marks). Returns the stack bottom (upper) address.
pointer	**Parameter Type**: String (***without*** quotation marks). Returns the stack pointer of the task.

Return Value Type: Hex value.