The CMX-RTX C-SPY plugin

Introduction to the CMX-RTX Debugger Plugin for the IAR Embedded Workbench C-SPY Debugger

This document describes the IAR C-SPY Debugger plugin for the CMX-RTX RTOS.

The CMX-RTX RTOS awareness plugin is delivered and installed as a part of the ARM® IAR Embedded Workbench™ IDE. For instructions on how to install and use the plugin, see the CMX documentation. CMX Systems, Inc., can be contacted via **www.cmx.com**.

To be able to use the plugin, you must do this: Start the ARM® IAR Embedded Workbench™ and choose **Project>Options**. Select the **C-SPY** category and choose **CMX-RTX** from the **RTOS support** combo box. Click **OK**.

Introduction to the plugin

The plugin introduces the following elements in the C-SPY user interface.

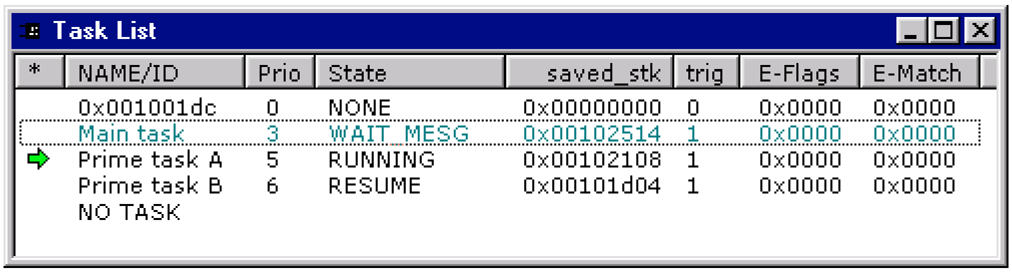
- In the View menu, it installs a number of entries corresponding to the various types of CMX-specific windows that can be opened by the plugin.
- In the **Breakpoints** dialog box, available by choosing **Edit>Breakpoints**, a new option is enabled for making standard breakpoints task-specific.
- It installs a new menu, named **CMX**, with entries for various RTOS-specific commands, in particular task-related stepping commands.
- It installs a new toolbar with buttons for commands from the **CMX** menu.

Windows

The RTOS plugin introduces seven new debugger windows.

The Task List Window

This is arguably the single most important window of the RTOS plugin.



*	NAME/ID	Prio	State	saved_stk	trig	E-Flags	E-Match
	0x001001dc	0	NONE	0x00000000	0	0x0000	0x0000
	Main task	3	WAIT_MESG	0x00102514	1	0x0000	0x0000
➡	Prime task A	5	RUNNING	0x00102108	1	0x0000	0x0000
	Prime task B	6	RESUME	0x00101d04	1	0x0000	0x0000
	NO TASK						

This window shows a list of all tasks created by the current application (by calls to `K_Task_Create`) and some items pertaining to their current state. The currently active task is indicated by an arrow in the first column (and typically by a state of `RUNNING` in the **State** column). The order of the tasks is that of the `cmx_tcb` array. If the `task_name` array is defined, those names will be used in the **Name/Id** column.

You can examine a particular task by double-clicking on the corresponding row in the window. All debugger windows (Watch, Locals, Register, Call Stack, Source, Diassembly etc) will then show the state of the program from the point of view of the task in question. A task selected in this way is indicated in the Task window by a different color (for the moment, a subdued blue color).

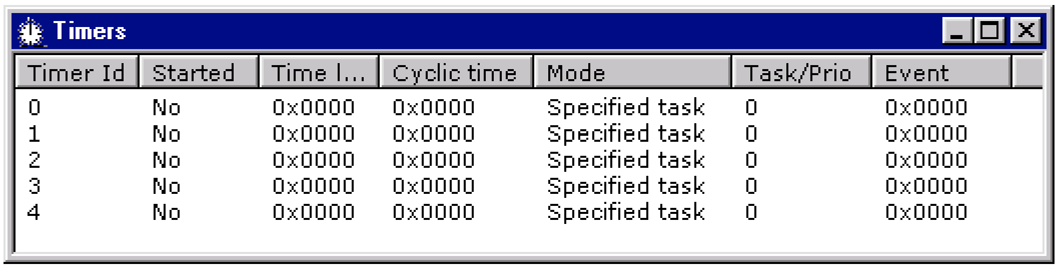
The last row of the Task window is always `NO TASK`. Double-clicking on this row makes the debugger show the state of the program as it currently is (that is, as it would be shown *without* an RTOS plugin), in effect always following the active task. Occasionally, such as when the pc is inside scheduler code, the arrow will point to the `NO TASK` entry.

Note that if a task has been selected by double-clicking, the debugger will show the state of that particular task until another task (or `NO TASK`) is selected, even if execution is performed by or in another task. For example, if task A is currently active (`RUNNING`) and you double-click on task B, which is `READY`, you will see information about the suspended task B. If you now perform a single-step by pressing F10, the active task (A) will perform a single-step, but since you are focused on task B, not much will actually visibly change.

Inspector Windows

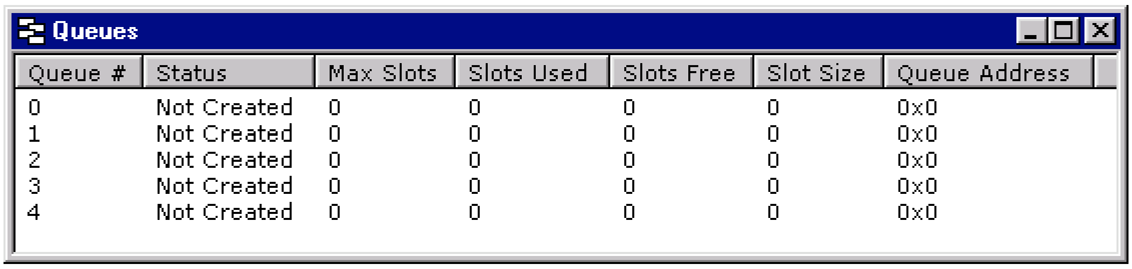
The six other windows display RTOS status information of various types. These windows are formatted but passive displays of various internal RTOS data structures, except for the System Variables window, which can also be used to modify the values of the shown variables.

Timers


Timers

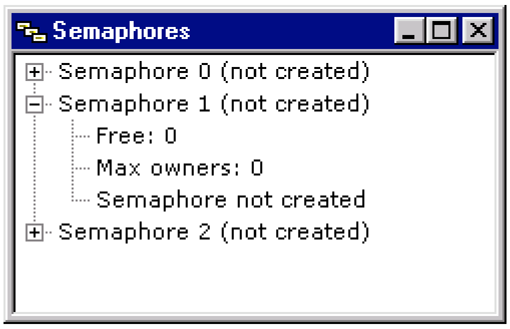
Timer Id	Started	Time l...	Cyclic time	Mode	Task/Prio	Event
0	No	0x0000	0x0000	Specified task	0	0x0000
1	No	0x0000	0x0000	Specified task	0	0x0000
2	No	0x0000	0x0000	Specified task	0	0x0000
3	No	0x0000	0x0000	Specified task	0	0x0000
4	No	0x0000	0x0000	Specified task	0	0x0000

Queues


Queues

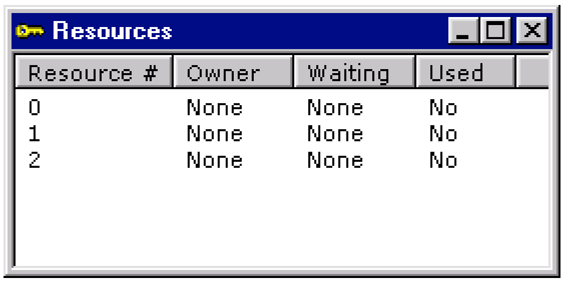
Queue #	Status	Max Slots	Slots Used	Slots Free	Slot Size	Queue Address
0	Not Created	0	0	0	0	0x0
1	Not Created	0	0	0	0	0x0
2	Not Created	0	0	0	0	0x0
3	Not Created	0	0	0	0	0x0
4	Not Created	0	0	0	0	0x0

Semaphores


Semaphores

```
⊞ Semaphore 0 (not created)
⊟ Semaphore 1 (not created)
    Free: 0
    Max owners: 0
    Semaphore not created
⊞ Semaphore 2 (not created)
```

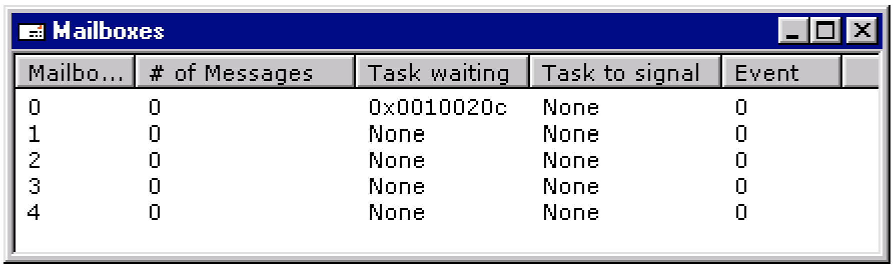
Resources



Resources

Resource #	Owner	Waiting	Used
0	None	None	No
1	None	None	No
2	None	None	No

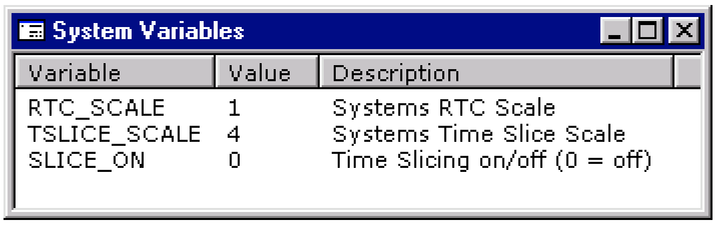
Mailboxes



Mailboxes

Mailbo...	# of Messages	Task waiting	Task to signal	Event
0	0	0x0010020c	None	0
1	0	None	None	0
2	0	None	None	0
3	0	None	None	0
4	0	None	None	0

System Variables

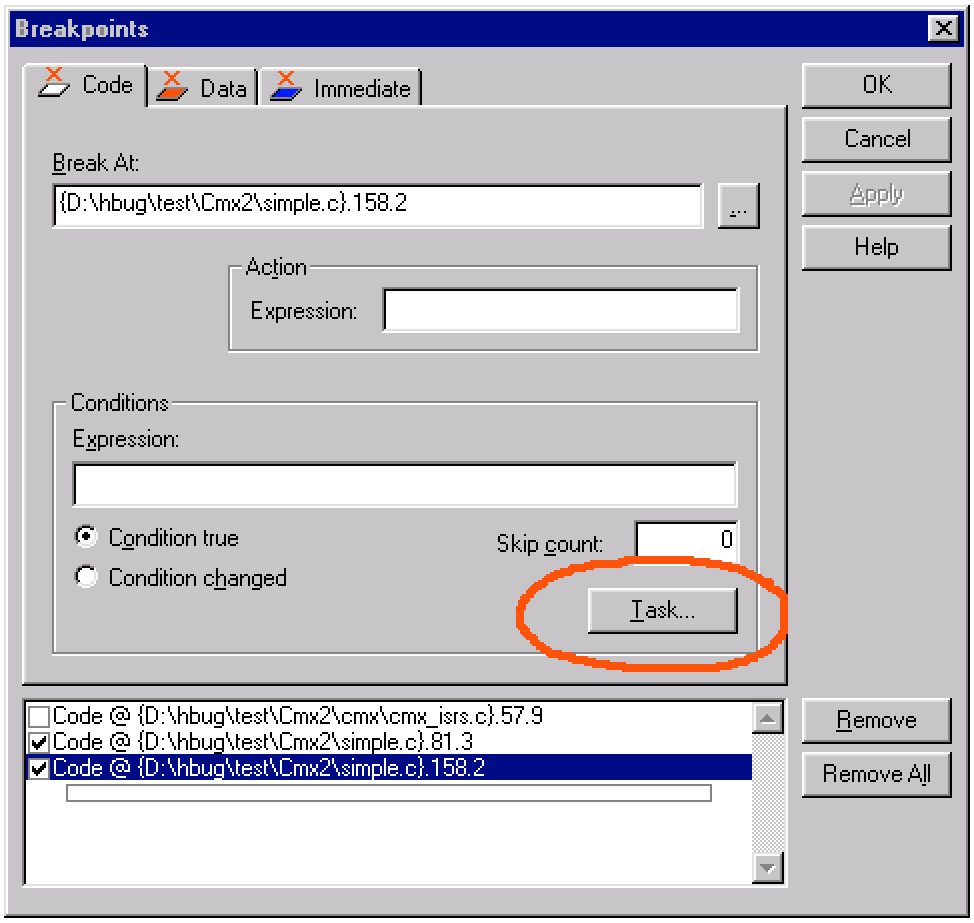


System Variables

Variable	Value	Description
RTC_SCALE	1	Systems RTC Scale
TSLICE_SCALE	4	Systems Time Slice Scale
SLICE_ON	0	Time Slicing on/off (0 = off)

Breakpoints

The presence of the RTOS plugin enables a task condition for all standard breakpoints, as shown below:



Break At: {D:\hbug\test\Cmx2\simple.c}.158.2

Action

Expression:

Conditions

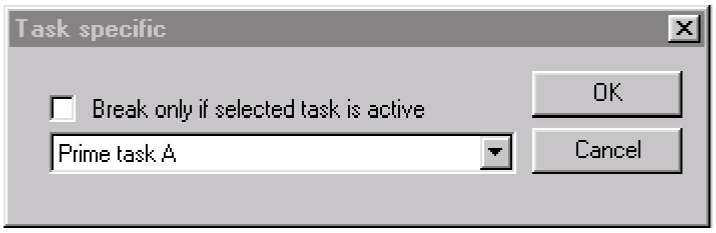
Expression:

(● Condition true) ○ Condition changed

Skip count: 0

☐ Code @ {D:\hbug\test\Cmx2\cmx\cmx_isrs.c}.57.9
☑ Code @ {D:\hbug\test\Cmx2\simple.c}.81.3
☑ Code @ {D:\hbug\test\Cmx2\simple.c}.158.2

Clicking the **Task…** button brings up the following dialog box:



☐ Break only if selected task is active

Prime task A

You can make a breakpoint task-specific by clicking the check-box and selecting a task from the drop-down list.

Note: The drop-down list only shows tasks which have been created at the time.

Note also: If the code at the breakpoint is only ever executed by one specific task, there is no need to make the breakpoint task-specific.

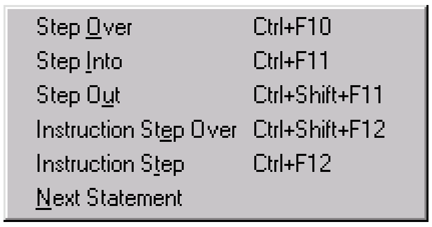
Stepping

If more than one task can execute the same code, there is a need both for task-specific breakpoints and for task-specific stepping.

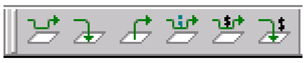
For example, consider some utility function, called by several different tasks. Stepping through such a function to verify its correctness can be quite confusing without task-specific stepping. Standard stepping usually works as follows (slightly simplified): When you invoke a step command, the debugger computes one or more locations where that step will end, sets corresponding temporary breakpoints and simply starts execution. When execution hits one of the breakpoints, they are all removed and the step is finished.

Now, during that brief (or not so brief) execution, basically anything can happen in an application with multiple tasks. In particular, a task switch may occur and *another* task may hit one of the breakpoints before the original task does. It may appear that you have performed a normal step, but now you are watching another task. The other task could have called the function with another argument or be in another iteration of a loop, so the values of local variables could be totally different.

Hence the need for task-specific stepping. The step commands on the **CMX** menu and on the corresponding toolbar behave just like the normal stepping commands, but they will make sure that the step doesn't finish until the *original* task reaches the step destination. This is the **CMX** menu:



And this is the **CMX** toolbar:



Important note: In the standard debugger menu, there are no **Instruction Step Over** and **Instruction Step** commands. This is because the standard **Step Over** and **Step Into** commands are context sensitive, stepping by statement and function call when a source window is active, and stepping by instruction when the Disassembly window is active. The RTOS stepping commands are unfortunately *not* context sensitive; you must choose which kind of step to perform.