The ThreadX C-SPY plugin

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

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

The ThreadX 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 ThreadX documentation. Express Logic, Inc., can be contacted via **www.expresslogic.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 Debugger** category and choose **ThreadX** from the **RTOS support** combo box. Click **OK**.

Introduction to the plugin

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

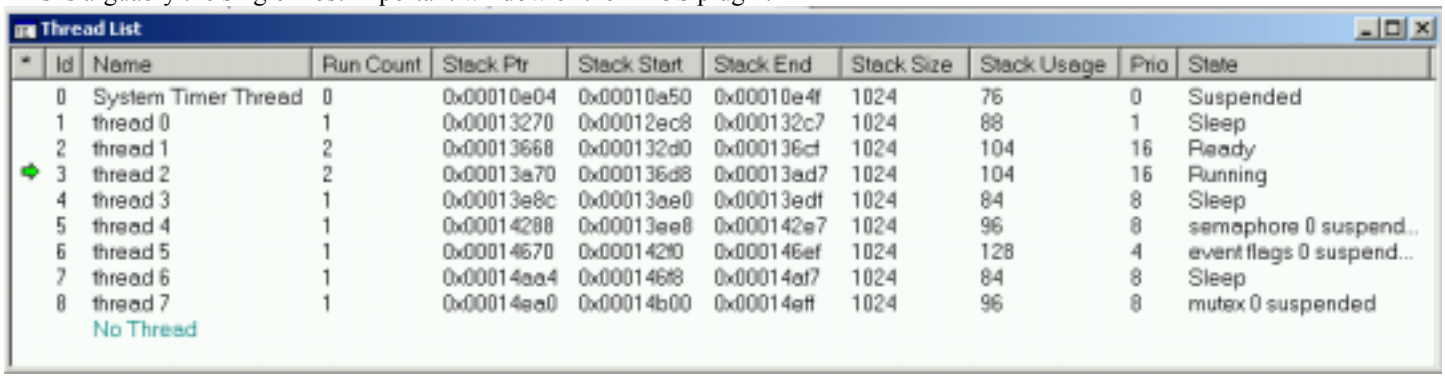
In the **View** menu, it installs a number of entries corresponding to the various types of ThreadX-specific windows that can be opened by the plugin.
In the **Breakpoints** dialog box, available by choosing **Edit>Breakpoints**, a new option is enabled for making standard breakpoints thread-specific.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

Windows

The RTOS plugin introduces seven new debugger windows. You can right-click in most of the windows to display a context menu where you can change display format (hexadecimal/decimal). If you select the **Color changes** command, all window updates are highlighted.

The Thread List Window

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



*	Id	Name	Run Count	Stack Ptr	Stack Start	Stack End	Stack Size	Stack Usage	Prio	State
	0	System Timer Thread	0	0x00010e04	0x00010a50	0x00010e4f	1024	76	0	Suspended
	1	thread 0	1	0x00013270	0x00012ec8	0x000132c7	1024	88	1	Sleep
	2	thread 1	2	0x00013668	0x000132d0	0x000136cf	1024	104	16	Ready
➡	3	thread 2	2	0x00013a70	0x000136d8	0x00013ad7	1024	104	16	Running
	4	thread 3	1	0x00013e8c	0x00013ae0	0x00013edf	1024	84	8	Sleep
	5	thread 4	1	0x00014288	0x00013ee8	0x000142e7	1024	96	8	semaphore 0 suspend...
	6	thread 5	1	0x00014670	0x000142f0	0x000146ef	1024	128	4	event flags 0 suspend...
	7	thread 6	1	0x00014aa4	0x000146f8	0x00014af7	1024	84	8	Sleep
	8	thread 7	1	0x00014ea0	0x00014b00	0x00014eff	1024	96	8	mutex 0 suspended
		No Thread								

This window shows a list of all threads created by the current application (by calls to `tx_thread_create`) and some items pertaining to their current state. The currently active thread is indicated by an arrow in the first column (and typically by a state of `RUNNING` in the **State** column). The order of the threads is that of the `_tx_thread_created_ptr`.

You can examine a particular thread 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 thread in question. A thread selected in this way is indicated in the Thread window by a different color (for the moment, a subdued blue color).

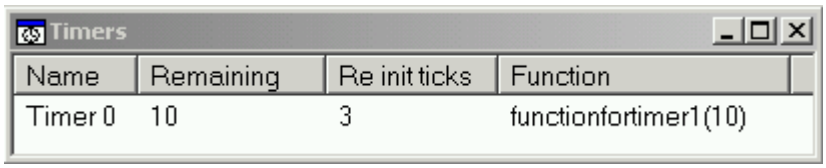
The last row of the Thread window is always `NO TASK`. Double-clicking on this row makes the debugger show the state of the program as it currently is (that is, as it would be shown *without* an RTOS plugin), in effect always following the active task.

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.

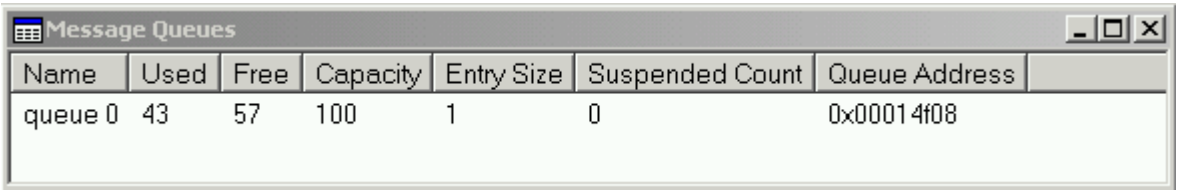
Inspector Windows

The seven other windows display RTOS status information of various types. These windows are formatted but passive displays of various internal RTOS data structures.

Timers



Queues

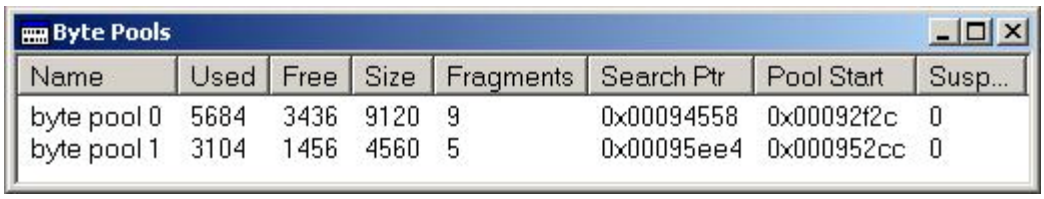


Semaphores

Mutexes

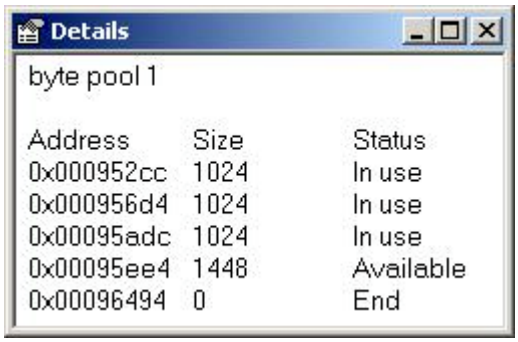
Event Flag Groups

Byte Pools



It is possible to get a detailed description on a specific byte pool. Right-click on a row and select the **Show Details** command from the context menu. The Details window is displayed. You can double-click any other row to select another byte pool while the Details window is open.

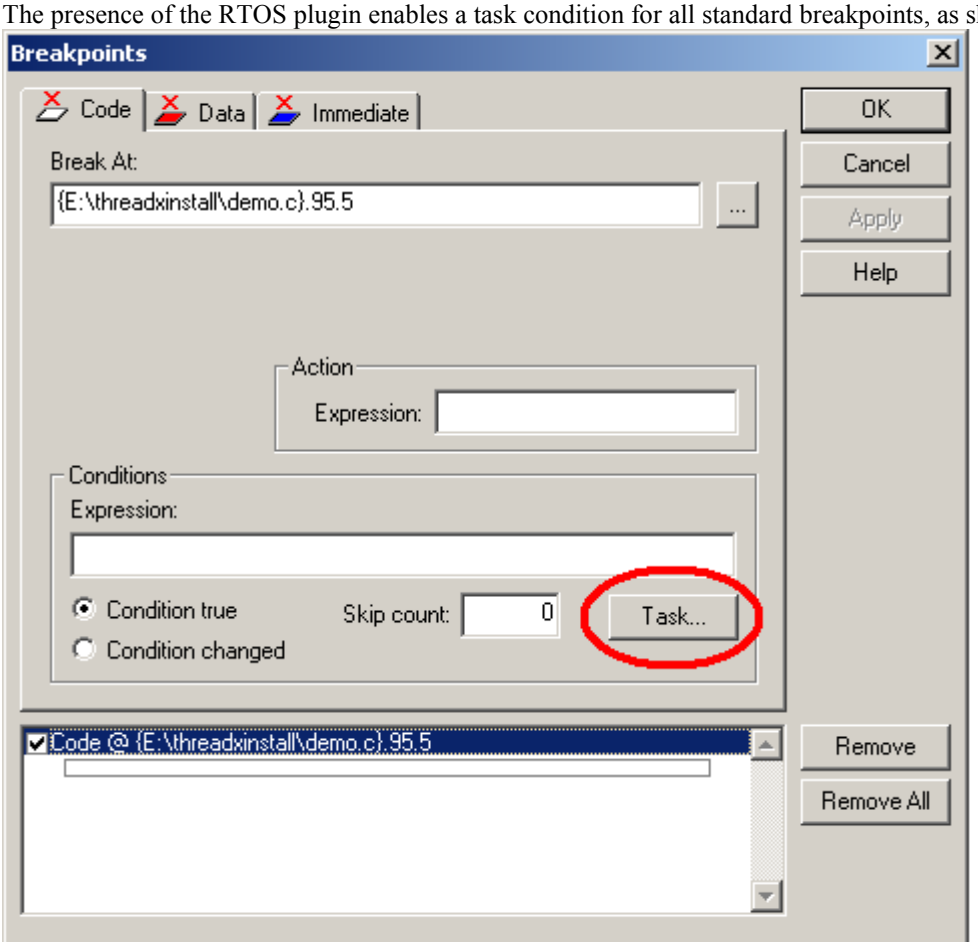
Details Window



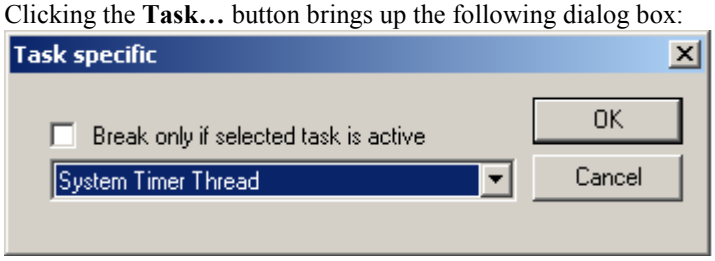
Block Pools

Breakpoints

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



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



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

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

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

Stepping

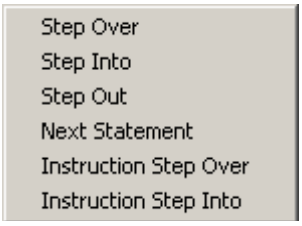
If more than one task can execute the same code, there is a need both for task-specific breakpoints and for task-specific stepping.

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, there is a need for task-specific stepping. The step commands on the **ThreadX** menu and on the corresponding toolbar behave just like the normal stepping commands, but they will make sure that the step does not finish until the *original* task reaches the step destination. This is the **ThreadX** menu:



And this is the **ThreadX** 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.