



C-SPY plugin

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

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 IAR Embedded Workbench™ IDE. The plugin provides extensive system information about all ThreadX resources, including threads, timers, queues, semaphores, mutexes, event flags, block pools, and byte pools. In addition, the plugin provides execution profile and performance metric information, when enabled by the developer.

For more information regarding the plugin, please visit Express Logic's website via www.expresslogic.com or send a request to info@expresslogic.com.

Enabling the Plugin

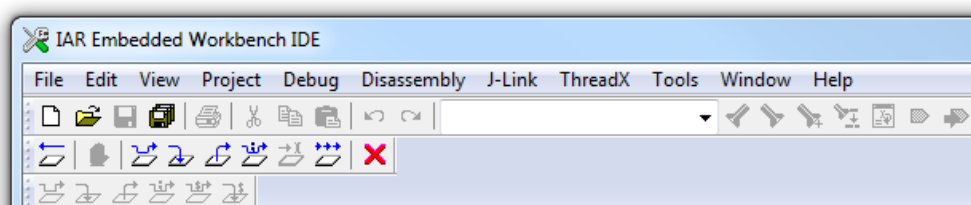
Enabling the ThreadX kernel awareness plugin is easy; simply follow these steps:

1. Start IAR Embedded Workbench™
2. Choose **Project -> Options**
3. Select **Debugger**
4. Select **Plugins** tab
5. Select **ThreadX** from the plugin list

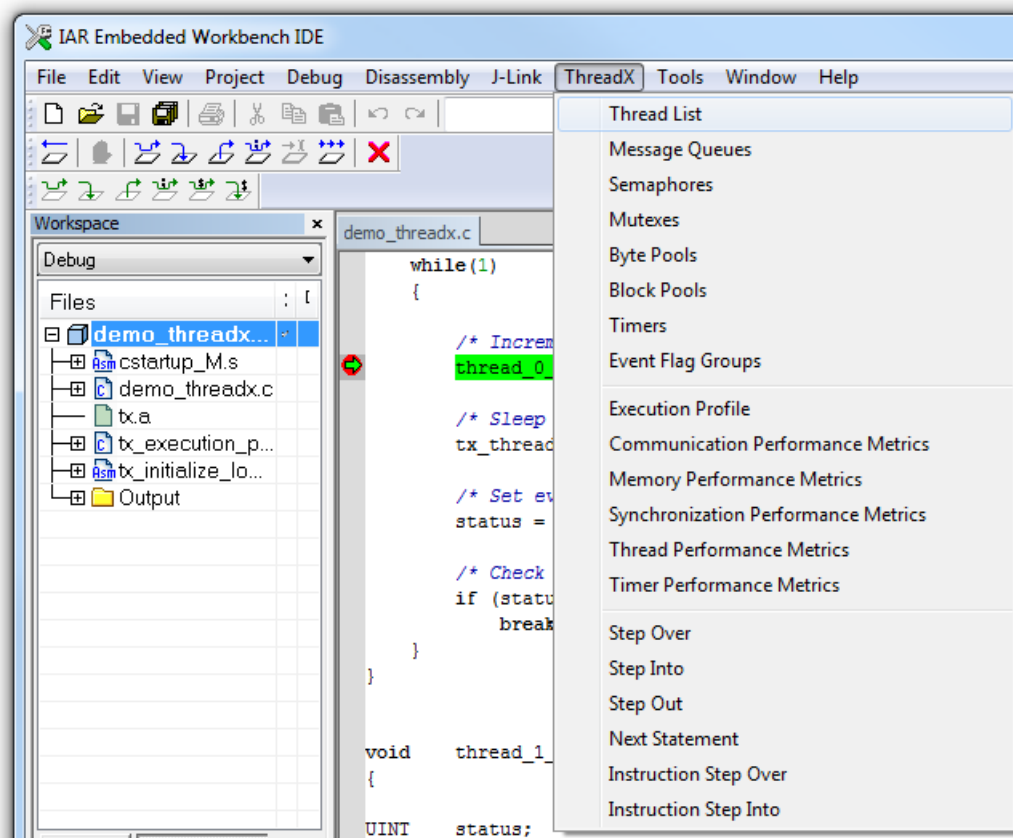
The plugin is now enabled and will be available on the next debug session of this project.

Introduction to the ThreadX plugin

The plugin introduces several elements in the C-SPY user interface. First, when the debugger is started the **ThreadX** menu item is visible, as shown:



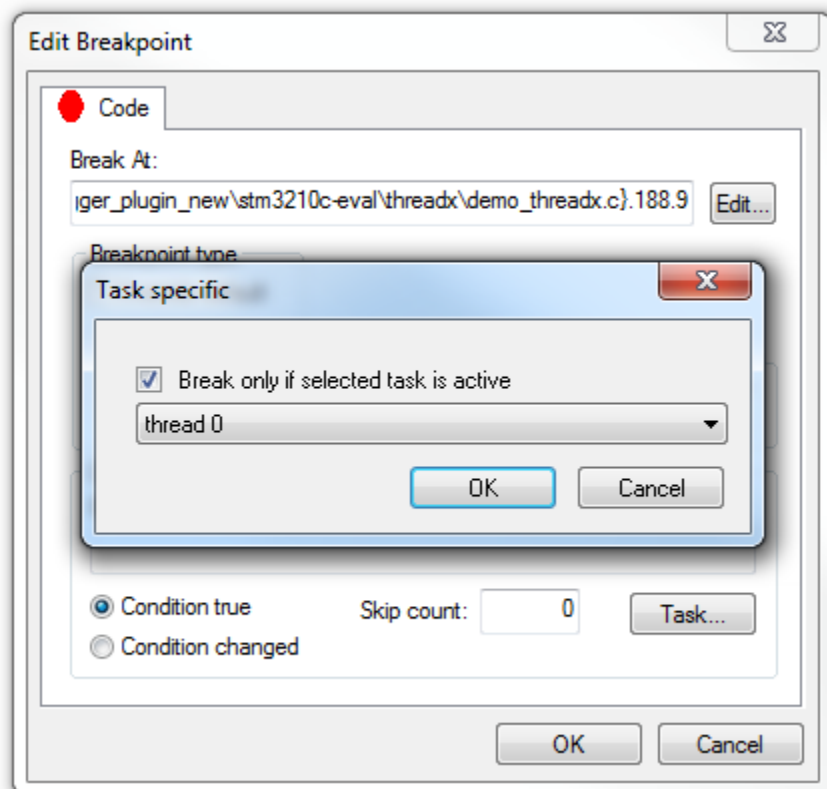
Selecting the **ThreadX** menu item yields the following pull-down menu, which contains selections for viewing various ThreadX information as well as thread-specific stepping control. The following shows the ThreadX pull-down menu:



From this menu, information is available for ThreadX threads, message queues, semaphores, mutexes, byte pools, block pools, timers and event flag groups. In addition, execution profiling and various performance metric information is available.

Thread-specific Breakpoints

Thread-specific breakpoints are available via a right-click on an existing breakpoint and selecting the **Edit Breakpoint** option. Once selected, the following dialog may be used to make the breakpoint thread-specific:



In this example, the breakpoint will only stop the processor if **thread 0** is executing at this location in the program.

Thread-specific Stepping

Thread-specific stepping is available from the main **ThreadX** menu and the thread-specific toolbar icons as shown:



If more than one thread can execute the same code, there is a need both for thread-specific breakpoints and for thread-specific stepping. For example, consider some utility function, called by several different threads. Stepping through such a function to verify its correctness can be quite confusing without thread-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 threads. In particular, a thread switch may occur and *another* thread may hit one of the breakpoints before the original thread does. It may appear that you have performed a normal step, but now you are watching another thread. The other thread 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 thread-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* thread reaches the step destination.

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 ThreadX stepping commands unfortunately are *not* context sensitive; you must choose which kind of step to perform.

ThreadX Display Windows

The ThreadX plugin introduces 14 additional debugger windows. You can right-click in most of the windows to enable/disable **Color changes** in that window. When Color changes are enabled (the default mode) changes from the last execution are highlighted in the color **red**.

The Thread List Window

The **Thread List Window** is arguably the single most important window of the ThreadX plugin. This window shows a list of all currently created threads in the application (by calls to `tx_thread_create`) and a series of 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 threads are listed in order of their creation.

The following shows the **Thread List Window** for the standard ThreadX demonstration. The example shows that **thread 0** is the currently executing thread. All the fields in red have changed since the last run/step command.

* ID	Name	Priority	State	Run Count	Stack Ptr	Stack Start	Stack End	Stack Size	Max Stack Usage
0	thread 0	1	Running	9	0x200013b8	0x20000fd0	0x200013cf	1024	112
1	thread 1	16	Ready	275	0x20001758	0x200013d8	0x200017d7	1024	128
2	thread 2	16	Ready	278	0x20001b88	0x200017e0	0x20001bdf	1024	128
3	thread 3	8	semaphore 0 suspended	40	0x20001f70	0x20001be8	0x20001fe7	1024	120
4	thread 4	8	Ready	40	0x20002380	0x20001ff0	0x200023ef	1024	120
5	thread 5	4	Ready	8	0x20002770	0x200023f8	0x200027f7	1024	136
6	thread 6	8	mutex 0 suspended	40	0x20002b88	0x20002800	0x20002bff	1024	120
7	thread 7	8	Ready	40	0x20002f98	0x20002c08	0x20003007	1024	120
	No Thread								

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, Disassembly 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 (a subdued blue color).

The last column of the display shows the maximum stack usage. As this number approaches the stack size, the greater the likelihood of a stack overflow.

The last row of the **Thread List Window** is always **No Thread**. 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* the ThreadX plugin), in effect always following the active thread.

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

The Execution Profile Window

The **Execution Profile Window** show exactly where the processing is taking place in the application. The percentages of time for which the system is idle, in interrupt processing, and in thread processing are shown. In addition, the percentage of execution time of each thread is also displayed. The information for this display requires the application and ThreadX to be built with execution profiling enabled, which requires use of the **Execution Profiling Kit**.

Please contact Express Logic to receive the **Execution Profile Kit (EPK)** and for help with any questions you might have in enabling execution profiling. The ThreadX plugin automatically detects the presence of execution profiling if it has been enabled. If it is not enabled, this window does not appear.

The following shows the execution profile of the standard ThreadX demonstration:

Execution	Total Time	Execution %	Individual Threads	Total Time	Execution %
Idle System	0	0.00%	thread 0	19737	0.08%
Interrupt	162439	0.68%	thread 1	11241920	46.72%
All Threads	23900631	99.32%	thread 2	12295982	51.10%
			thread 3	72581	0.30%
Total	24063070	100%	thread 4	72124	0.30%
			thread 5	13355	0.06%
			thread 6	92657	0.39%
			thread 7	92275	0.38%

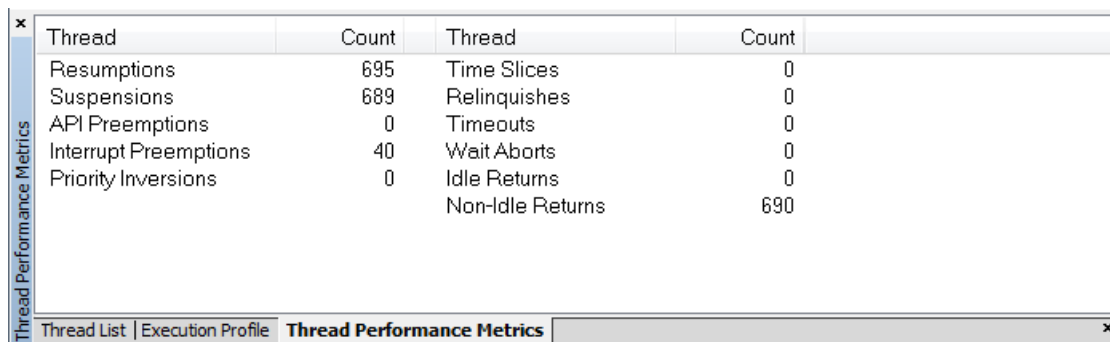
In the standard ThreadX demonstration, **thread 1** and **thread 2** execute continuously, and thus consume 46.72% and 51.10% of the execution, respectively. Since one of these threads is always ready, there is no idle time, as shown by 0%. The timer interrupt in this simple demonstration represents 0.68% of the execution.

With this feature, developers are able to more fully analyze their application to determine how much processing is still available as well as where most of the processing is taking place. From this information a developer can determine if there are enough available processor cycles to handle maximum system loading requirements (safety margin) or to add additional application functionality. Alternatively, if there is an excessive amount of available cycles, the developer may choose to lower the frequency of the processor to reduced power consumption. Finally, the information provides an excellent roadmap for optimization, since it shows exactly where the processing is taking place.

The Thread Performance Metrics Window

The **Thread Performance Metrics Window** shows a variety of internal ThreadX counters for events such as thread resumptions, suspensions, preemptions via an API call, interrupt preemptions, priority inversions, time-slices, relinquishes, timeouts, wait abort API calls, idle and non-idle returns from a thread. This information is optionally gathered by ThreadX and is enabled by building the ThreadX library with **TX_THREAD_ENABLE_PERFORMANCE_INFO** defined.

An example display of thread performance metrics for the standard ThreadX demonstration is shown as follows:



The screenshot shows a window titled "Thread Performance Metrics" with a close button (x) in the top right corner. The window contains a table with two columns: "Thread" and "Count". The table is divided into two sections by a vertical line. The left section lists: Resumptions (695), Suspensions (689), API Preemptions (0), Interrupt Preemptions (40), and Priority Inversions (0). The right section lists: Time Slices (0), Relinquishes (0), Timeouts (0), Wait Aborts (0), Idle Returns (0), and Non-Idle Returns (690). At the bottom of the window, there are three tabs: "Thread List", "Execution Profile", and "Thread Performance Metrics" (which is currently selected). A small "x" button is also visible in the bottom right corner of the window frame.

Thread	Count	Thread	Count
Resumptions	695	Time Slices	0
Suspensions	689	Relinquishes	0
API Preemptions	0	Timeouts	0
Interrupt Preemptions	40	Wait Aborts	0
Priority Inversions	0	Idle Returns	0
		Non-Idle Returns	690

This example shows that there were 695 thread resumptions and 689 thread suspensions. There were 40 interrupt preemptions associated with higher-priority thread 0 waking up from its sleep every 10 ticks. There are also 0 idle returns, meaning that in this example, there is always at least one thread ready for execution, i.e., no idle time.

From this information, a developer might see an excessive number of thread preemptions, which is effectively a full context switch. Seeing this, the developer may re-evaluate the assigned thread priorities in order to reduce the number of thread preemptions and thus reduce overhead.

The Synchronization Performance Metrics Window

The **Synchronization Performance Metrics Window** shows a variety of counters associated with the ThreadX synchronization objects, e.g., semaphores, mutexes, and event flag groups. This information is optionally gathered by ThreadX and is enabled by building the ThreadX library with the following defined:

```
TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO
TX_MUTEX_ENABLE_PERFORMANCE_INFO
TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
```

An example display of synchronization performance metrics for the standard ThreadX demonstration is shown as follows:

Mutex	Count	Event Flags	Count	Semaphore	Count
Puts	78	Sets	8	Puts	39
Gets	81	Gets	8	Gets	41
Suspensions	40	Suspensions	8	Suspensions	40
Timeouts	0	Timeouts	0		0
Priority Inversions	0				
Priority Inheritance	0				

Thread List | Execution Profile | Communication Performance Metrics | **Synchronization Performance Metrics** | x

This example shows 81 attempts to obtain a mutex and 78 mutex releases. There are 40 thread suspensions associated with attempts to obtain a mutex that is not available. It is noteworthy that the mutex operations exceed that of any other synchronization object. From this information, the developer may explore ways to reduce the number of mutex operations in order to reduce overhead.

The Communication Performance Metrics Window

The **Communication Performance Metrics Window** shows a variety of internal ThreadX counters associated with communication message queues, including total messages sent, messages received, empty suspensions, full suspensions, queue full errors, and timeouts associated with queue full or empty conditions. This information is optionally gathered by ThreadX and is enabled by building the ThreadX library with **TX_QUEUE_ENABLE_PERFORMANCE_INFO** defined.

An example display of queue performance metrics for the standard ThreadX demonstration is shown as follows:

Queue	Count
Messages Sent	26213
Messages Received	26151
Empty suspensions	256
Full suspensions	257
Full Errors	0
Timeouts	0

Thread List | Execution Profile | **Communication Performance Metrics** | x

This example shows 26,213 messages sent and 26,151 messages received. There were 256 empty queue suspensions and 257 queue full conditions. There were no timeouts or queue full errors (attempts to send to a full queue without a suspension option). From this information, the developer may explore increasing queue depth in order to reduce the number for queue full suspensions.

The Timer Performance Metrics Window

The **Timer Performance Metrics Window** shows a variety of internal ThreadX counters associated with timers, including total activations, reactivations, deactivations, expirations, and internal adjustments. This information is optionally gathered by ThreadX and is enabled by building the ThreadX library with **TX_TIMER_ENABLE_PERFORMANCE_INFO** defined.

An example display of timer performance metrics for the standard ThreadX demonstration is shown as follows:

Timer	Count
Activations	1
Reactivations	20
Deactivations	0
Expirations	42
Internal Adjustments	0

This example shows 42 timer expirations. Only 1 timer was activated and this was a periodic timer that was reactivated 20 additional times. From this information, the developer may explore increasing timer periods to reduce the number of expirations.

The Memory Performance Metrics Window

The **Memory Performance Metrics Window** shows a variety of counters associated with the ThreadX memory management objects, e.g., variable-length byte pools and fixed-length block pools. This information is optionally gathered by ThreadX and is enabled by building the ThreadX library with the following defined:

**TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO**

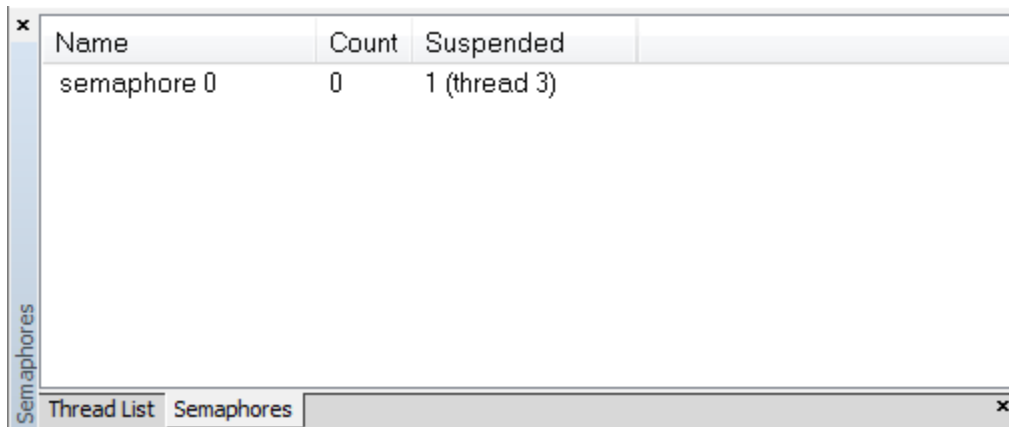
An example display of byte and block pool performance metrics for the standard ThreadX demonstration is shown as follows:

Byte Memory	Count	Block Memory	Count
Allocations	10	Allocations	1
Deallocations	0	Deallocations	1
Suspensions	0	Suspensions	0
Timeouts	0	Timeouts	0
Internal Merges	0		
Internal Fragments	10		
Blocks Searched	10		

This example shows 10 byte pool allocations, 10 byte pool fragments, and 10 byte pool searches. In addition, there is 1 block pool allocation and 1 block pool release. From this information, the developer may explore adjusting the size of the byte or block pool if there are excessive pool suspensions.

The Semaphores Window

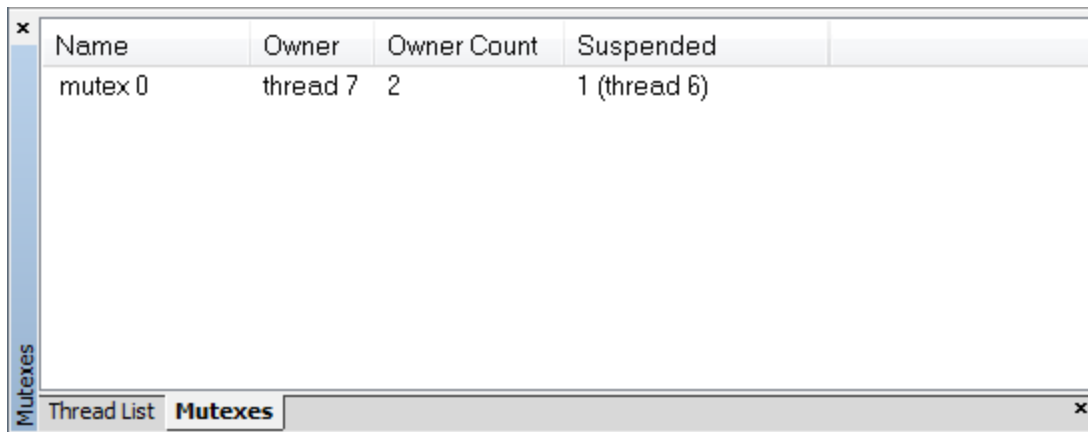
The **Semaphores Window** shows the status of the currently created semaphores in the application. The information includes the current semaphore count and the number of threads suspended on the semaphore along with the name of the first thread suspended. The following is an example of the semaphore display for the standard ThreadX demonstration:



Name	Count	Suspended
semaphore 0	0	1 (thread 3)

The Mutexes Window

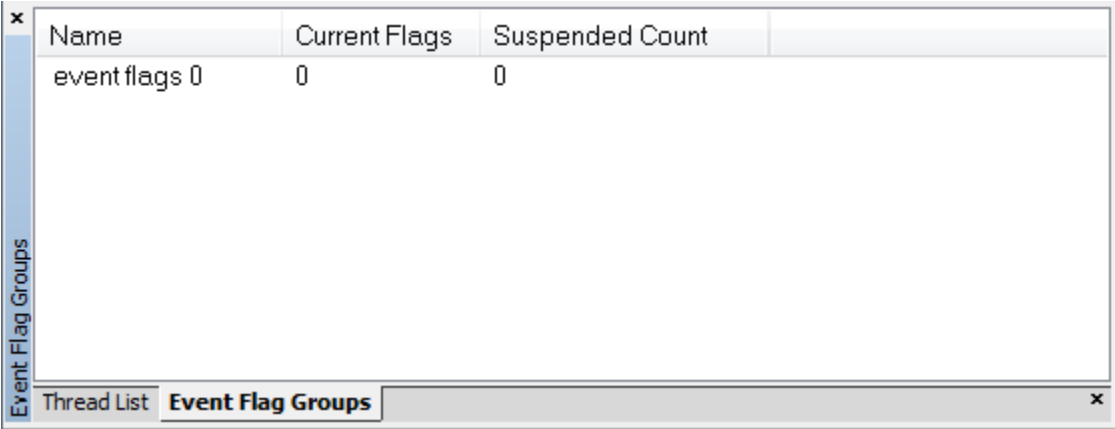
The **Mutexes Window** shows the status of the currently created mutexes in the application. The information includes the current mutex owner, the ownership count and the number of threads suspended on the mutex along with the name of the first thread suspended. The following is an example of the mutex display for the standard ThreadX demonstration:



Name	Owner	Owner Count	Suspended
mutex 0	thread 7	2	1 (thread 6)

The Event Flag Groups Window

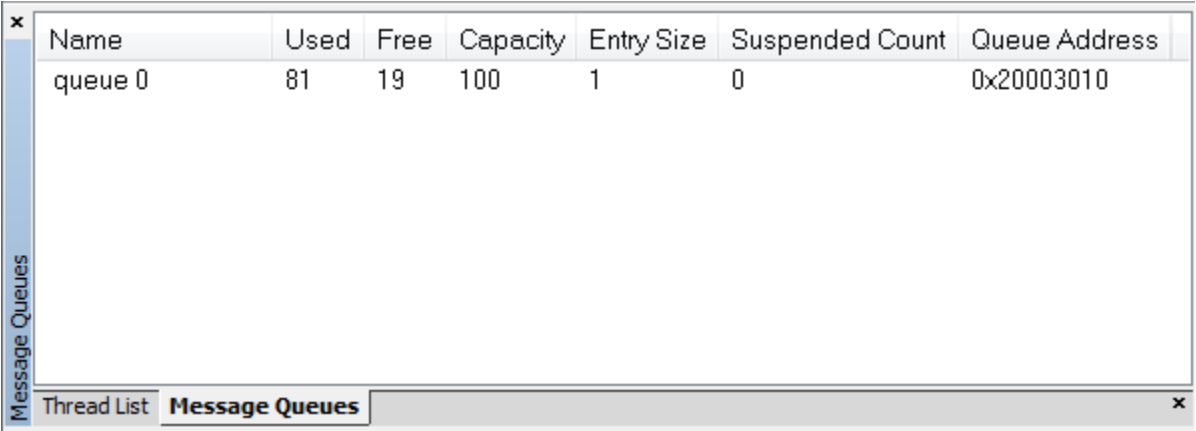
The **Event Flag Groups Window** shows the status of the currently created event flags in the application. The information includes the current event flags set and the number of threads suspended on the event flag group along with the name of the first thread suspended. The following is an example of the event flag display for the standard ThreadX demonstration:



Name	Current Flags	Suspended Count
event flags 0	0	0

The Message Queues Window

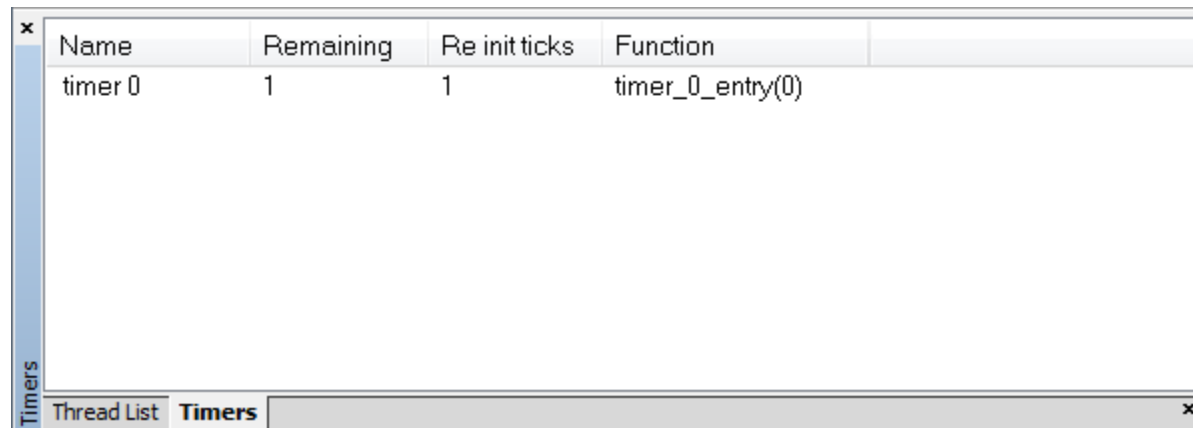
The **Message Queues Window** shows the status of the currently created message queues in the application. The information includes the capacity of the queue, number of free entries, number of used entries, the size of each entry (in terms of 32-bit words), and the number of threads suspended on the message queue along with the name of the first thread suspended. The following is an example of the message queue display for the standard ThreadX demonstration:



Name	Used	Free	Capacity	Entry Size	Suspended Count	Queue Address
queue 0	81	19	100	1	0	0x20003010

The Timers Window

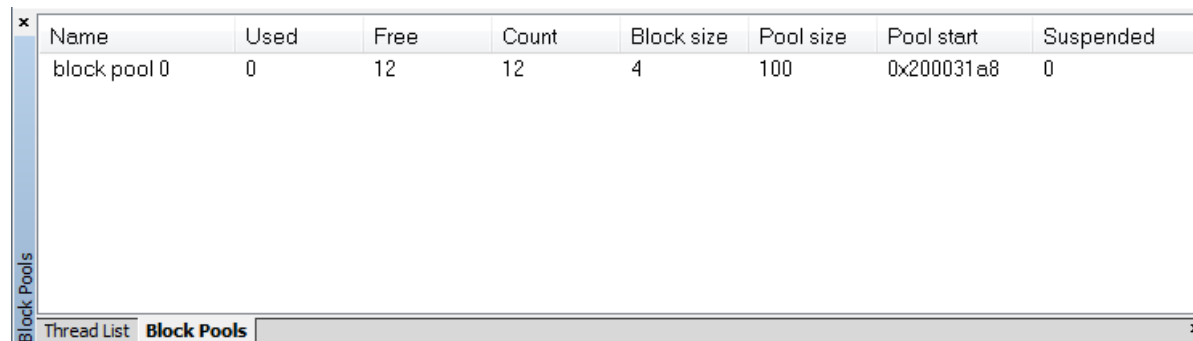
The **Timers Window** shows the status of the currently created timers in the application. The information includes the remaining timer ticks before expiration, the re-initialization ticks (for periodic timers), and the entry function pointer. The following is an example of the timer display for the standard ThreadX demonstration:



Name	Remaining	Re init ticks	Function
timer 0	1	1	timer_0_entry(0)

The Block Pools Window

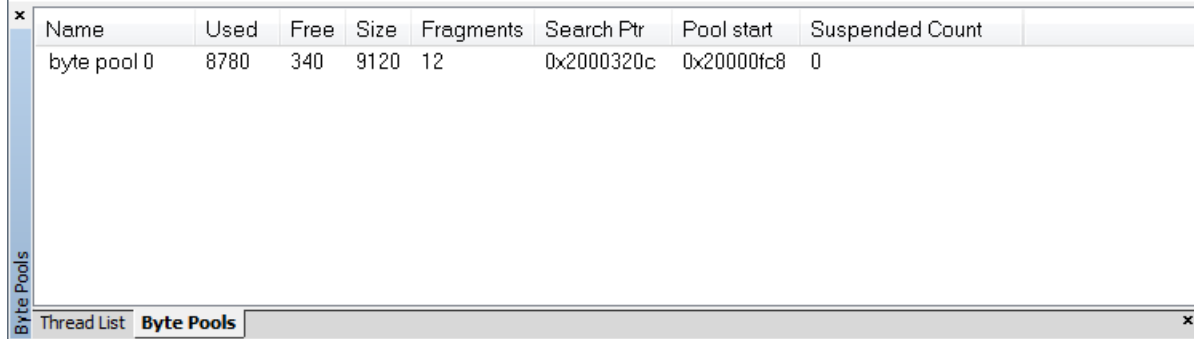
The **Block Pools Window** shows the status of the currently created block pools in the application. The information includes the number of allocated (used) blocks, free blocks, block size (in bytes), total pool size, and the number of threads suspended on the block pool along with the name of the first thread suspended. The following is an example of the block pool display for the standard ThreadX demonstration:



Name	Used	Free	Count	Block size	Pool size	Pool start	Suspended
block pool 0	0	12	12	4	100	0x200031a8	0

The Byte Pools Window

The **Byte Pools Window** shows the status of the currently created byte pools in the application. The information includes the amount of allocated (used) memory, free memory, fragments, total pool size, and the number of threads suspended on the byte pool along with the name of the first thread suspended. The following is an example of the byte pool display for the standard ThreadX demonstration:



Name	Used	Free	Size	Fragments	Search Ptr	Pool start	Suspended Count
byte pool 0	8780	340	9120	12	0x2000320c	0x20000fc8	0