MULTI: Developing for ThreadX

Green Hills Software
30 West Sola Street
Santa Barbara, California 93101
USA
Tel: 805-965-6044
Fax: 805-965-6343
www.ghs.com

LEGAL NOTICES AND DISCLAIMERS

PubID: threadx-544152
Branch: http://toolsvc/branches/release-branch-70-bto
Date: October 21, 2015

Contents

Preface

Contents

This preface discusses the purpose of the manual, the MULTI documentation set, and typographical conventions used.

About This Book

This book describes settings, filenames, and procedures that apply specifically to developing with MULTI for ThreadX. For more comprehensive documentation of MULTI features, consult the other books in the documentation set, as described in “The MULTI Document Set” on page ix.

This book is divided into two parts:

- *Part I: Using MULTI with ThreadX* explains how to debug ThreadX applications and describes specialized windows for viewing ThreadX kernel components. See Part I. Using MULTI with ThreadX on page 1.
- *Part II: Using the MULTI EventAnalyzer for ThreadX* describes how to collect and view event logging data in the EventAnalyzer. See Part II. Using the MULTI EventAnalyzer for ThreadX on page 71.

Note

New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your installation directory for release notes, **README** files, and other supplementary documentation.

MULTI for ThreadX

ThreadX is a high-performance real-time embedded kernel developed by Express Logic, Inc. The MULTI Integrated Development Environment works seamlessly with ThreadX to provide detailed kernel-aware and thread-aware debugging for developers, including full C, C++, and Embedded C++ source and assembly-language debugging.

All eight ThreadX kernel components are recognized by MULTI:

- Threads
- Message queues
- Semaphores
- Mutexes
- Event flags groups
- Memory block pools
- Memory byte pools
- Application timers

Each kernel component type has at least two associated MULTI windows: a *list window* that summarizes all created kernel components of that component type and an *information window* that shows detailed information about a specific component. The thread, block pool, and byte pool kernel components also have additional associated windows that provide further information. All of these windows are described in detail in the following chapters.

The MULTI Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Provides an introduction to the MULTI Integrated Development Environment and leads you through a simple tutorial.
- *MULTI: Licensing* — Describes how to obtain, install, and administer MULTI licenses.
- *MULTI: Managing Projects and Configuring the IDE* — Describes how to create and manage projects and how to configure the MULTI IDE.

- *MULTI: Building Applications* — Describes how to use the compiler driver and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to configure connections to your target.
- *MULTI: Debugging* — Describes how to set up your target debugging interface for use with MULTI and how to use the MULTI Debugger and associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Scripting* — Describes how to create MULTI scripts. Also contains information about the MULTI-Python integration.

For a comprehensive list of the books provided with your MULTI installation, see the **Help → Manuals** menu accessible from most MULTI windows.

All books are available in one or more of the following formats:

- Print.
- Online help, accessible from most MULTI windows via the **Help → Manuals** menu.
- PDF, available in the **manuals** subdirectory of your MULTI or compiler installation.

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Indication	Example
bold type	Filename or pathname	**C:\MyProjects**
	Command	**setup** command
	Option	**-G** option
	Window title	The **Breakpoints** window
	Menu name or menu choice	The **File** menu
	Field name	**Working Directory:**
	Button name	The **Browse** button
italic type	Replaceable text	**-o *filename***
	A new term	A task may be called a *process* or a *thread*
	A book title	*MULTI: Debugging*
`monospace` type	Text you should enter as presented	Type `help` *`command_name`*
	A word or words used in a command or example	The **wait** [-global] command blocks command processing, where `-global` blocks command processing for all MULTI processes.
	Source code	`int a = 3;`
	Input/output	`> print Test` `Test`
	A function	`GHS_System()`
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	**debugbutton** [*name*]...
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	`> print Test` `Test`
pipe (|) (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	**call** *func* | *expr*

Convention	Indication	Example
square brackets ([ ]) (in command line instructions)	Optional argument, command, option, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	`.macro` *`name`* `[`*`list`*`]`

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-*option*]... *filename*

The formatting of this command indicates that:

- The command **gxyz** should be entered as shown.
- The option *`-option`* should either be replaced with one or more appropriate options or be omitted.
- The word *`filename`* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

Part I

Using MULTI with ThreadX

Chapter 1

Running MULTI for ThreadX

Contents

This chapter provides a basic overview of how to use MULTI to debug ThreadX applications.

ThreadX and Green Hills Tools Compatibility

MULTI and the Green Hills Compilers are compatible with ThreadX 5.5 and later.

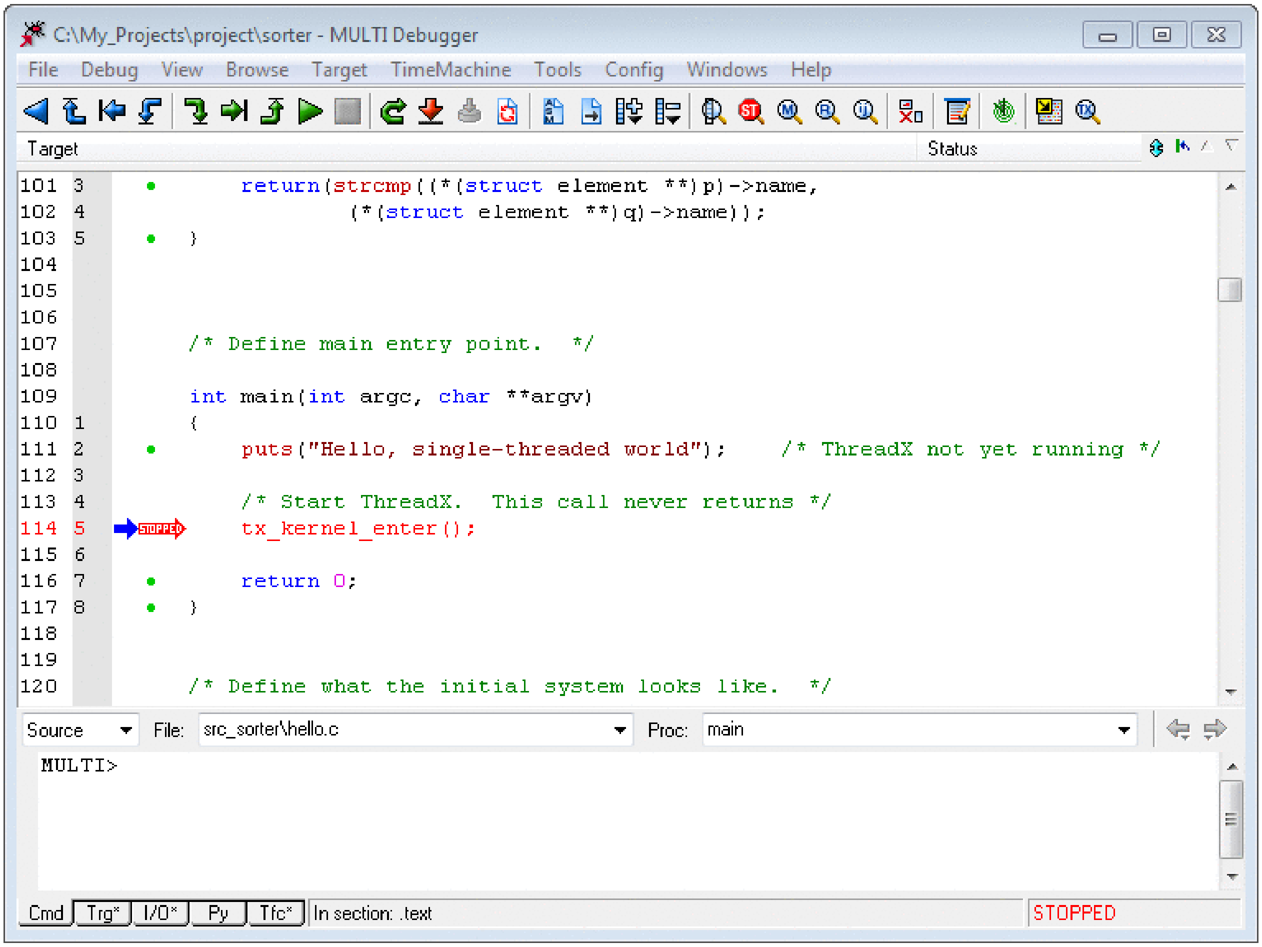
For a project built with ThreadX, if you customize system libraries, remove the following files from your **libsys.gpj** subproject:

- **ind_thrd.c**
- **ind_lock.c**
- **ind_except.c**

ThreadX-specific versions of the routines contained in these files are included with ThreadX.

Debugging ThreadX Applications

When you debug a ThreadX application with MULTI, a ThreadX icon ( ) appears in the MULTI Debugger toolbar:



Click the ThreadX icon to open the **ThreadX Information** window, the main control window for MULTI kernel-aware debugging for ThreadX. See "The ThreadX Information Window" on page 6 for more information about using this window.

To learn more about MULTI for ThreadX, use MULTI to create a product demonstration program. You must have a licensed copy of ThreadX installed on your system to build a ThreadX program.

Manipulating ThreadX Windows

The following information applies to all ThreadX windows except the **Thread List** window.

- Clicking the blue **Freeze** button (•) located near the top right corner of the window freezes the window and changes the button into a snowflake ().

Clicking the snowflake makes the window active again. An active window is updated each time the target is stopped. See “Performance Issues” on page 10 for more information about the **Freeze** button.

- Entering **Ctrl**+**d** when a window is in view freezes the window and creates a duplicate, active copy of it.
- Clicking a button for a particular component displays more details about that component.
- Double-clicking any item in a list displays information about it. If no useful information exists for an item, double-clicking the item may have no effect.

Alignment Restrictions

It is best to ensure 4-byte alignment and sizes for most component options that refer to addresses or size in memory. To ensure correct alignment, ThreadX pads certain size parameters to be multiples of 4 bytes or adjusts beginning or ending pointers to be 4-byte aligned.

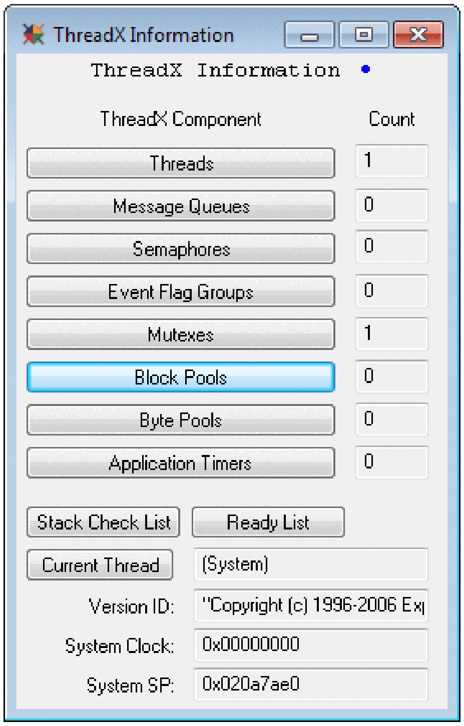
This alignment restriction can sometimes explain differences between what is specified when a component is created and what is displayed when it is viewed. For example, a memory block pool created with a pool size of 258 bytes is not able to make use of any more than 256 bytes. Similarly, creating a block pool with a block size of 10 bytes results in an actual block size of 12 bytes.

Timeout Values

References to thread time slices and suspended thread timeout values, as well as to application timer values inside ThreadX windows, refer only to the timeout values contained in the underlying data structures. These entries do not necessarily count down as time elapses. Counting down every one of these values on each timer tick would compromise the real-time performance of ThreadX.

The ThreadX Information Window

The **ThreadX Information** window is the main control window for MULTI kernel-aware debugging for ThreadX. To open this window, click the ThreadX button ( ) in the MULTI Debugger window.



The **ThreadX Information** window shows useful system parameters such as individual component counts, the name of the current thread, the version ID string, and the status of the system clock and stack pointer. Buttons in the window open other windows that contain lists for each component type or detailed information about the current thread. Each field and button is described more specifically in the table below.

Threads	Shows the number of created threads in the system, which corresponds to the system variable `_tx_thread_created_count`. Clicking the **Threads** button opens the **Thread List** window (see “The Thread List Window” on page 14).
Message Queues	Shows the number of created message queues in the system, which corresponds to the system variable `_tx_queue_created_count`. Clicking the **Message Queues** button opens the **Queue List** window (see “The Queue List Window” on page 28).
Semaphores	Shows the number of created semaphores in the system, which corresponds to the system variable `_tx_semaphore_created_count`. Clicking the **Semaphores** button opens the **Semaphore List** window (see “The Semaphore List Window” on page 34).

Event Flag Groups	Shows the number of created event flags groups in the system, which corresponds to the system variable `_tx_event_flags_created_count`. Clicking the **Event Flag Groups** button opens the **Event Flags List** window (see "The Event Flags List Window" on page 44).
Mutexes	Shows the number of created mutexes in the system, which corresponds to the system variable `_tx_mutex_created_count`. Mutex objects are available starting with ThreadX version 4.0. Clicking the **Mutexes** button opens the **Mutex List** window (see "The Mutex List Window" on page 38).
Block Pools	Shows the number of created block pools in the system, which corresponds to the system variable `_tx_block_pool_created_count`. Clicking the **Block Pools** button opens the **Block Pool List** window (see "The Block Pool List Window" on page 50).
Byte Pools	Shows the number of created byte pools in the system, which corresponds to the system variable `_tx_byte_pool_created_count`. Clicking the **Byte Pools** button opens the **Byte Pool List** window (see "The Byte Pool List Window" on page 58).
Application Timers	Shows the number of created application timers in the system, which corresponds to the system variable `_tx_timer_created_count`. Clicking the **Application Timers** button opens the **Timer List** window (see "The Timer List Window" on page 66).
Stack Check List	Opens the **Thread Stack Check List** window, which displays stack information for all threads in the system (see "The Thread Stack Check List Window" on page 24).
Ready List	Opens the **Thread Ready List** window, which shows a list of all threads that are at the same priority level as the currently executing thread and are ready to execute (see "The Thread Ready List Window" on page 17).
Current Thread	Shows the current executing thread, which corresponds to the thread pointed to by the system variable `_tx_thread_current_ptr`. If the system is not within a thread, then `(System)` is displayed. Clicking the **Current Thread** button opens the **Current Thread Information** window (a **Thread Information** window on the thread that was executing when the system was stopped). See "The Current Thread Information Window" on page 23 and "The Thread Information Window" on page 19 for more information.
Version ID	Shows the version ID string of the system. This value corresponds to the string pointed to by the system variable `_tx_version_id`.

System Clock	Shows in timer ticks the system clock status. This value corresponds to the system variable `_tx_timer_system_clock`.
System SP	Shows the value of the system stack pointer. This value corresponds to the system variable `_tx_thread_system_stack_ptr`.

Checking Thread Stack Usage

Stack overflow is a common problem that MULTI helps diagnose.

The **Stack Use** field in any thread window shows how much stack is currently in use by each thread and how much stack space is available for each thread. This information can help you to identify threads that are using more stack space than anticipated and to adjust their stack sizes to guard against overflow before problems occur.

MULTI for ThreadX also provides peak stack checking. You can check peak stack use for a single thread by viewing the **Stack Check Information** window (see "The Stack Check Information Window" on page 25) or for all threads by viewing the **Thread Stack Check List** window (see "The Thread Stack Check List Window" on page 24). In either case, MULTI displays the peak stack usage as determined by the highest point in the stack that has changed since the thread was created. Peak stack use checking occurs by executing code on the target itself, which is usually much faster than uploading large portions of target memory to the host.

Configuring Stack Use Checking

MULTI Debugger-based stack use checking is enabled by default in ThreadX. In ThreadX versions 3 and 4, stack use checking can be disabled by compiling `tx_tc.c` with the preprocessor symbol TX_DISABLE_STACK_CHECKING defined. In ThreadX version 5, stack use checking can be disabled by rebuilding the ThreadX library with the TX_DISABLE_STACK_FILLING configuration option.

When stack use checking is enabled, the `tx_thread_create` service fills a thread's stack with an `0xEF` data pattern that is used by the MULTI Debugger to calculate stack usage. This function can be bypassed , which results in threads being created more quickly.

ThreadX version 5 also contains a separate run-time stack checking feature. That feature can be enabled or disabled separately from the MULTI Debugger-based stack use checking.

Analyzing ThreadX Memory Allocation

MULTI for ThreadX contains enhanced views of memory block pools and byte pools to help developers find problems with dynamic memory allocation.

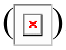
To view this enhanced information, click **In Use** in a **Block Pool Information** window or **Byte Pool Information** window. This opens a **Contents** window that lists the pool's memory blocks or fragments by their location in memory and indicates whether each is `In Use` or `Available`. For byte pools, the window also displays the size of each byte pool fragment. These memory contents windows make it easy to observe the effects of dynamic memory allocation and to detect the causes of byte pool fragmentation. See “The Block Pool Contents Window” on page 55 or “The Byte Pool Contents Window” on page 62 for more information.

The MULTI EventAnalyzer

The MULTI EventAnalyzer can display ThreadX event information that allows users of ThreadX to analyze the complex, real-time interactions occurring in their target systems. For more information about configuring and using the EventAnalyzer, please refer to the second part of this book, “Using the MULTI EventAnalyzer for ThreadX”.

Performance Issues

All ThreadX windows except **Stack Check** windows are automatically updated each time the target is halted, hits a breakpoint, or stops for any other reason. Thus, the more ThreadX windows you have displayed, the more target data is uploaded to the host system each time the target stops, and the slower your debugging performance may be. To maximize debugging performance, close any unnecessary ThreadX windows.

Another way to speed debugging is to use the blue **Freeze** button () located near the upper right-hand corner of each ThreadX window to selectively freeze windows that do not need updating. Click the button to freeze the window. The button is replaced by a snowflake (). Click the snowflake to make the window active again. An active window updates every time the target stops; frozen windows are not updated with target data. To force an update of all active windows, use the **update** command in the MULTI Debugger.

Chapter 2

Threads

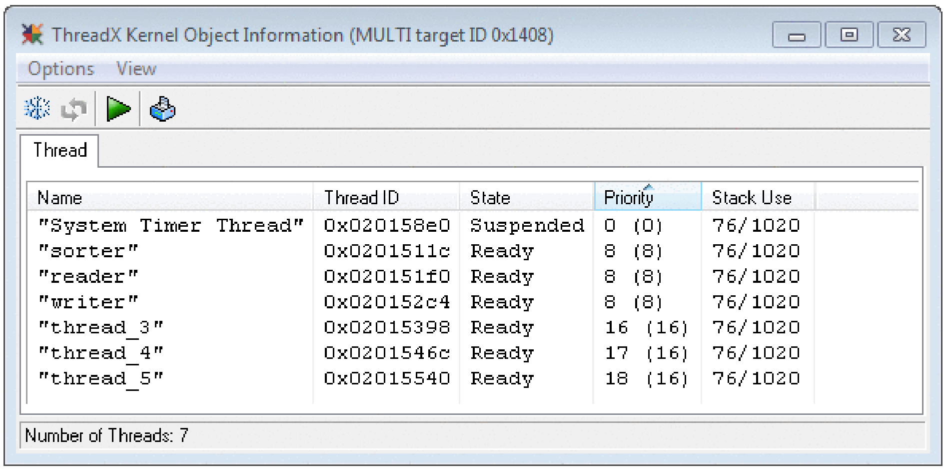
Contents

This chapter describes windows that display detailed information about the threads in your application.

The Thread List Window

The **Thread List** window shows a list of all threads in the kernel, in the order in which they were created.

To display this window, click the **Threads** button in the main **ThreadX Information** window.



The list is generated by following a linked list, starting with the thread pointed to by the global variable `_tx_thread_created_ptr` and continuing with the tx_thread_created_next field of each thread control block TX_THREAD. A total of `_tx_thread_created_count` threads are shown.

When a thread is deleted, it disappears from the **Thread List** window and from the Debugger's target list.

Multithreaded Debugging

The **Thread List** window can be used for freeze-mode multithreaded debugging. To display a thread in the MULTI Debugger, single-click it in the Debugger's target list, or double-click it in the **Thread List** window.

For further information about working with the **Thread List** window, refer to the documentation about freeze-mode debugging and OS-awareness in the *MULTI: Debugging* book.

Contents of the Thread List Window

The **Thread List** window displays up to seven columns: **Name**, **Thread ID**, **State**, **Priority**, **Stack Use**, **Run Count**, and **Suspended On**. Each of these columns is described below.

Note

Not all columns are shown by default; right-click the column header to open a menu that will allow you to display or hide any of the available columns.

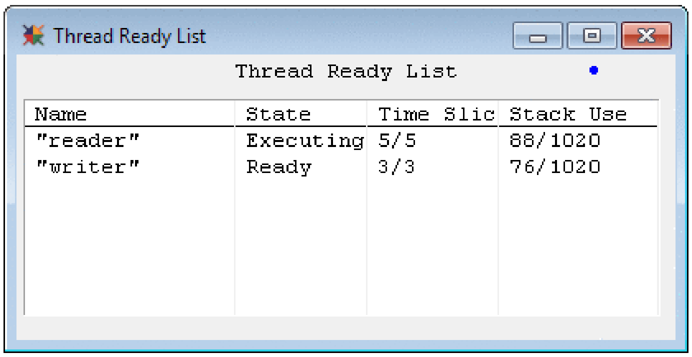
Name	Displays the name of the thread, as given in the call to `tx_thread_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the thread control block TX_THREAD. This entry corresponds to the tx_thread_name field of TX_THREAD.
Thread ID	Displays the address of the thread control block TX_THREAD, as given in the call to `tx_thread_create`.
State	Indicates the current execution state of the thread. This entry corresponds to the tx_thread_state field of TX_THREAD. The thread can be in one of five states: • `Executing` — The thread is executing. • `Ready` — The thread is ready and will execute when it is the highest priority thread. • `Suspended` — The thread cannot run because it is waiting. Threads can wait for time, message queues, event flags, semaphores, mutexes, and memory, or can be placed in a suspended state upon thread creation. • `Terminated` — The thread was terminated by a `tx_thread_terminate` call. • `Completed` — The thread has returned from its entry function.

Priority	Gives information about the priority of the thread. The first number is the priority level of the thread, which corresponds to the tx_thread_priority field in TX_THREAD. The second number, in parentheses, is the preemption threshold of the thread and corresponds to the tx_thread_preempt_threshold field in TX_THREAD.
Stack Use	Indicates the amount of stack currently in use by the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used and is derived from the difference between the tx_thread_stack_end field in TX_THREAD and the current thread stack pointer. The second number indicates the total amount of stack space allocated to the thread and is the difference between the tx_thread_stack_end and tx_thread_stack_start fields in TX_THREAD.
Run Count	Indicates how many times the thread has been scheduled. When this field is increasing, the thread is being scheduled and run. A run counter that stays the same may indicate a thread that is unable to run for some reason. This field corresponds to the tx_thread_run_count field of TX_THREAD.
Suspended On	Indicates the type (`Queue`, `Semaphore`, `Mutex`, `Event Flags Group`, `Block Pool`, `Byte Pool`, `Sleep`, or `Suspend Call`) of the component on which the thread is suspended and its name (as given when that component was created; if a `0` (null pointer) was passed as the `name_ptr` argument, the address of the control block is displayed). The name portion of this field is derived from the appropriate name field of the component pointed to by the tx_thread_suspend_control_block field of TX_THREAD. The component type portion of this field is derived from the tx_thread_state field of TX_THREAD. This field shows `N/A` if the execution state is anything other than `Suspended`.

The Thread Ready List Window

The **Thread Ready List** window shows a list of all threads that are at the same priority level as the currently executing thread and are ready to execute.

To display this window, click the **Ready List** button in the **ThreadX Information** window.



The list is generated by following a linked list, starting with the thread pointed to by the global variable `_tx_thread_current_ptr` and continuing with the tx_thread_ready_next field of each thread control block TX_THREAD.

When a thread is deleted, it disappears from this list.

You can double-click any thread in the **Thread Ready List** to display a **Thread Information** window (see “The Thread Information Window” on page 19).

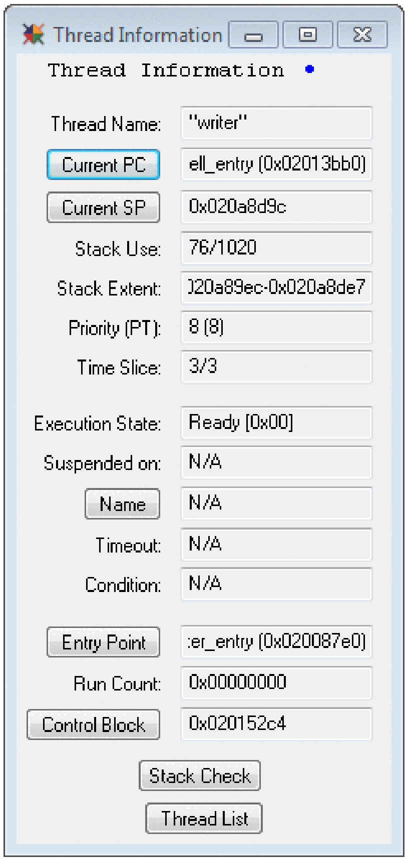
The **Thread Ready List** has four columns: **Name**, **State**, **Time Slice**, and **Stack Use**. Each of these is described below.

Name	Displays the name of the thread, as given in the call to `tx_thread_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the thread control block TX_THREAD. This entry corresponds to the tx_thread_name field of TX_THREAD.
State	Indicates the current execution state of the thread. The thread can be in one of two states: • `Executing` — The thread is executing. • `Ready` — The thread is ready and will execute when it is the highest priority thread.

Time Slice	Displays the timer ticks remaining and the timer ticks given to the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the remaining timer ticks in the slice and corresponds to the tx_thread_time_slice field of TX_THREAD. The second number indicates how many timer ticks the thread will receive when it is subsequently scheduled after it exhausts its current time slice. This number is the time_slice parameter passed to `tx_thread_create` and corresponds to the tx_thread_new_time_slice field of TX_THREAD.
Stack Use	Indicates the amount of stack currently in use by the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used and is derived from the difference between the tx_thread_stack_end field in TX_THREAD and the current thread stack pointer. The second number indicates the total amount of stack space allocated to the thread and is derived from the difference between the tx_thread_stack_end and tx_thread_stack_start fields in TX_THREAD.

The Thread Information Window

The **Thread Information** window shows detailed information about an individual thread.



To display this window, double-click a thread in the **Thread Ready List** window, right-click a thread in the **Thread List** window, or view a variable of type TX_THREAD in the MULTI Debugger window. The information in this window is derived from various fields within the thread control block TX_THREAD.

If a thread is deleted while a **Thread Information** window for it exists, the window does not disappear; the window continues to show the contents of the thread control block.

Each of the fields and buttons in the **Thread Information** window is described next.

Thread Name	Displays the name of the thread, as given in the call to `tx_thread_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_thread_name field of TX_THREAD.
Current PC	Gives the name of the function in which the thread is currently executing. If no debugging information is available for that location, this field may be displayed as an offset from a known label or as an address in hexadecimal format. This field is derived from one of the following: 1. The system program counter (PC), if the thread is currently executing. 2. A PC value as stored on the stack, if the thread is not currently executing. Clicking the **Current PC** button displays the current PC location.
Current SP	Identifies the current stack pointer of the thread, displayed as a hexadecimal address. For threads that are not currently executing, this field corresponds to the tx_thread_stack_ptr field of TX_THREAD. For the currently executing thread, this field displays the processor's stack pointer register. Clicking the **Current SP** button displays a memory view of the thread's stack.
Stack Use	Indicates the amount of stack currently in use by the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used and is derived from the difference between the tx_thread_stack_end field in TX_THREAD and the current thread stack pointer. The second number indicates the total amount of stack space allocated to the thread and is derived from the difference between the tx_thread_stack_end and tx_thread_stack_start fields in TX_THREAD.
Stack Extent	Displays the stack range as two hexadecimal addresses. This field is derived from the tx_thread_stack_start and tx_thread_stack_end fields in TX_THREAD.
Priority (PT)	Gives information about the priority of the thread. The first number is the priority level of the thread, which corresponds to the tx_thread_priority field in TX_THREAD. The second number, in parentheses, is the preemption threshold of the thread and corresponds to the tx_thread_preempt_threshold field in TX_THREAD.

Time Slice	Displays the timer ticks remaining and the timer ticks given to the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the remaining timer ticks in the slice and corresponds to the tx_thread_time_slice field of TX_THREAD. The second number indicates how many timer ticks the thread will receive when it is subsequently scheduled after it exhausts its current time slice. This number is the time_slice parameter passed to tx_thread_create and corresponds to the tx_thread_new_time_slice field of TX_THREAD.
Execution State	Indicates the current execution state of the thread. This entry is derived from the tx_thread_state field of TX_THREAD and by comparing the global variable `_tx_thread_current_ptr` with the address of TX_THREAD. The thread can be in one of five states: • `Executing` — The thread is executing. • `Ready` — The thread is ready and will execute when it is the highest priority thread. • `Suspended` — The thread cannot run because it is waiting. Threads can wait for time, message queues, event flags, semaphores, mutexes, and memory, or can be placed or created in a suspended state. • `Terminated` — The thread was terminated by a `tx_thread_terminate` call. • `Completed` — The thread has returned from its entry function.
Suspended on	Names the component type (`Queue`, `Semaphore`, `Mutex`, `Event Flags Group`, `Block Pool`, `Byte Pool`, `Sleep`, or `Suspend Call`) on which the thread is suspended. This field is derived from the tx_thread_state field of TX_THREAD and shows `N/A` if the execution state is anything other than `Suspended`.
Name	Indicates the name of the component on which the thread is suspended, as given when that component was created. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays the address of the control block. This field is derived from the appropriate name field of the component pointed to by the tx_thread_suspend_control_block field of TX_THREAD and shows `N/A` if the execution state is anything other than `Suspended`. Clicking the **Name** button displays a view of the component on which the thread is suspended.

Timeout	Indicates the number of ticks specified in whatever action caused the suspend. After these ticks elapse, the thread will no longer be suspended. If the suspension was caused by an attempt to access another kernel component, a suitable error value will be returned from the service call. This field shows `Forever` if `TX_WAIT_FOREVER` was specified as the wait value in the service call that caused the thread to be suspended. This field corresponds to the tx_timer_internal_remaining_ticks field of the tx_thread_timer structure within TX_THREAD and shows `N/A` if the execution state is anything other than `Suspended`.
Condition	Shows particular information about why a thread is suspended on a queue, event flags group, or byte pool. • For a queue, this field displays `Receive` or `Send`. • For an event flags group, this field shows the particular flags being requested, as well as TX_AND, TX_AND_CLEAR, TX_OR, or TX_OR_CLEAR, as appropriate. • For byte pool allocation requests, this field shows the number of bytes requested. This field is derived from the tx_thread_suspend_option and tx_thread_suspend_info fields of TX_THREAD, except in the case of queues, when it is derived from the tx_queue_enqueued field of TX_QUEUE. This field shows `N/A` if the thread is not suspended on a queue, event flags group, or byte pool access.
Entry Point	Gives the name of the function called upon thread startup. If no debugging information is available for that location, then this field may be displayed as an offset from a known label or as an address in hexadecimal format. This field corresponds to the tx_thread_entry_function field of TX_THREAD. Clicking the **Entry Point** button displays the entry point function.
Run Count	Gives a count of how many times a thread has been scheduled. When this field is increasing, the thread is being scheduled and run. A run count that stays the same may indicate a thread that is unable to run for some reason. This field corresponds to the tx_thread_run_count field of TX_THREAD.
Control Block	Shows the address of the thread control block, which is a variable of type TX_THREAD. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the thread control block. The **Data Explorer** window displays useful information that is not included in the **Thread Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).

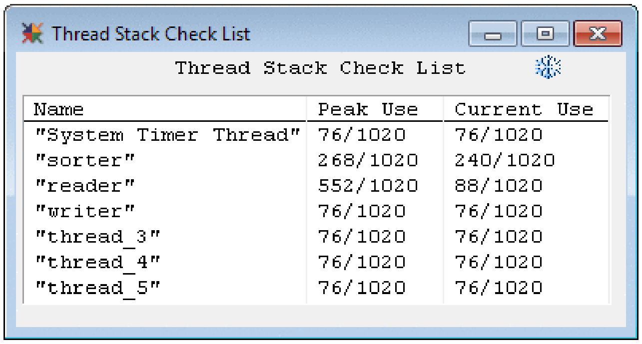
Stack Check	Displays the **Stack Check Information** window, which shows the peak stack usage of the thread. If stack checking is not enabled, this button has no effect. See “Checking Thread Stack Usage” on page 9 and “The Stack Check Information Window” on page 25 for more information about stack checking.
Thread List	Displays the **Thread List** window, which contains a list of all threads in the system (see “The Thread List Window” on page 14).

The Current Thread Information Window

The **Current Thread Information** window includes the same fields and buttons as the **Thread Information** window. The **Current Thread Information** window, however, displays information corresponding to the currently executing thread rather than a specifically selected thread. The information in this window is derived from the `_tx_thread_current_ptr` global variable. See “The Thread Information Window” on page 19 for a description of the fields in the **Current Thread Information** window.

The Thread Stack Check List Window

The **Thread Stack Check List** window shows all threads in the system together with their maximum stack usage, arranged in the order the threads were created. To open this window, click **Stack Check List** in the **ThreadX Information** window.



The list is generated by following a linked list, starting with the thread pointed to by the global variable `_tx_thread_created_ptr` and continuing with the tx_thread_created_next field of each thread control block TX_THREAD. A total of `_tx_thread_created_count` threads are shown.

This list is frozen immediately upon its creation because it causes code to be executed on the target system, which may not always be desirable. For information about refreshing this window, and all others, see “Performance Issues” on page 10. See “Checking Thread Stack Usage” on page 9 for more information about using this list.

From the **Thread Stack Check List** window, you can double-click any listed task to display a **Thread Information** window (see “The Thread Information Window” on page 19).

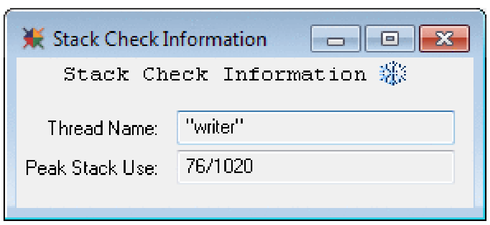
The **Thread Stack Check List** has three columns: **Name**, **Peak Use**, and **Current Use**. Each of these is described next.

Name	Displays the name of the thread, as given in the call to `tx_thread_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the thread control block TX_THREAD. This field corresponds to the tx_thread_name field of TX_THREAD.

Peak Use	Indicates the maximum amount of stack ever used by a thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used. The second number indicates the total amount of stack space allocated to the thread. This value is determined by examining the stack and finding the highest point on the stack that was changed from its original value of `0xef`.
Current Use	Gives the amount of stack currently in use by the thread. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used, and the second number indicates the total amount of stack space allocated to the thread.

The Stack Check Information Window

The **Stack Check Information** window shows the maximum stack usage of a thread.



To open this window, click the **Stack Check** button in the **Thread Information** window. The **Stack Check Information** window is frozen immediately upon its creation because it causes code to be executed on the target system, which may not always be desirable. For information about refreshing this window, and all others, see "Performance Issues" on page 10. See "Checking Thread Stack Usage" on page 9 for more information about stack checking.

If a thread is deleted while a **Stack Check Information** window for that thread exists, the window does not automatically disappear; the window continues to display stack check information.

The two fields of the **Stack Check Information** window are described below.

Thread Name	Gives the name of thread, as given in the call to tx_thread_create. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the thread control block TX_THREAD. This field corresponds to the tx_thread_name field of TX_THREAD.

Peak Stack Use	Indicates the maximum amount of stack ever used by the thread. This is determined by examining the stack and finding the highest point on the stack that was changed from its original value of `0xef`. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the amount of stack the thread has used, and the second number indicates the total amount of stack space allocated to the thread.

Chapter 3

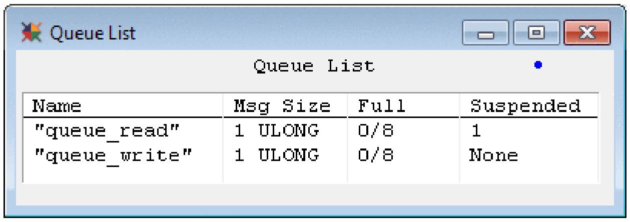
Message Queues

Contents

This chapter describes windows that display detailed information about the message queues in your application.

The Queue List Window

The **Queue List** window shows a list of all message queues in the system, arranged in the order in which they were created. To display this window, click the **Message Queues** button in the **ThreadX Information** window.



The information in the list is generated by following a linked list, starting with the message queue pointed to by the global variable `_tx_queue_created_ptr` and continuing with the tx_queue_created_next field of each queue control block TX_QUEUE. A total of `_tx_queue_created_count` message queues are shown.

When a queue is deleted, it is removed from this list.

You can double-click any message queue in the **Queue List** window to display a **Queue Information** window (see “The Queue Information Window” on page 29).

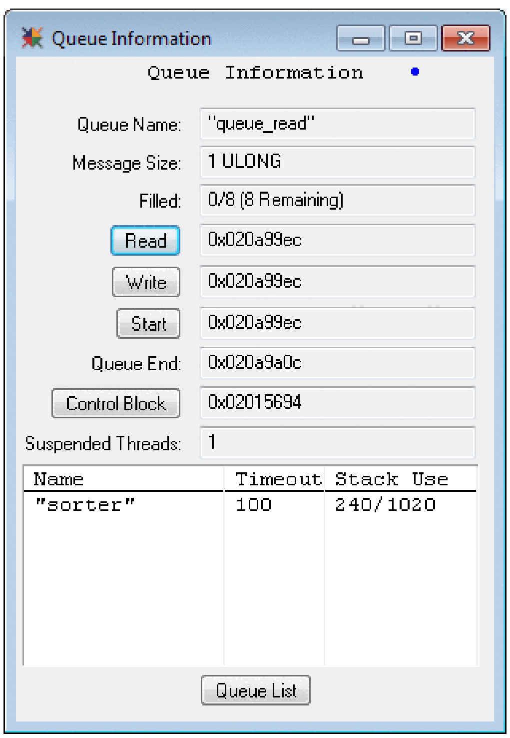
The **Queue List** window has four columns: **Name**, **Msg Size**, **Full**, and **Suspended**. Each of these is described below.

Name	Displays the name of the queue, as given in the call to tx_queue_create. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the queue control block TX_QUEUE. This entry corresponds to the tx_queue_name field of TX_QUEUE.
Msg Size	Indicates the size of each message in the queue. Message sizes range from one to sixteen 32-bit words (ULONGs). Valid message sizes are 1, 2, 4, 8, and 16 words. This entry corresponds to the tx_queue_message_size field of TX_QUEUE.

Full	Shows the number of messages currently stored in the queue awaiting a call to `tx_queue_receive`. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of messages currently stored in the queue, and the second number indicates the total number of messages. These numbers are derived from the tx_queue_enqueued and tx_queue_available_storage fields of TX_QUEUE.
Suspended	Shows the number of threads currently suspended on attempted accesses to the message queue, or displays `None` if no threads are suspended. This field corresponds to the tx_queue_suspended_count field of TX_QUEUE.

The Queue Information Window

The **Queue Information** window shows detailed information about an individual message queue.



To display this window, double-click any queue in the **Queue List** window, or view a variable of type TX_QUEUE in the MULTI Debugger window. The information in the **Queue Information** window is derived from various fields within the queue control block TX_QUEUE.

If a message queue is deleted while a **Queue Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the queue control block.

Each of the fields and buttons in the **Queue Information** window is described below.

Queue Name	Displays the name of the queue, as given in the call to `tx_queue_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_queue_name field of TX_QUEUE.
Message Size	Indicates the size of each message in the queue. Message sizes range from one to sixteen 32-bit words (ULONGs). Valid message sizes are 1, 2, 4, 8, and 16 words. This field corresponds to the tx_queue_message_size field of TX_QUEUE.
Filled	Shows the number of messages currently stored in the queue awaiting a call to `tx_queue_receive`. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of messages currently stored in the queue, and the second number indicates the total number of messages. The number of available messages (the total number of messages minus the number of messages stored in the queue) is also shown after this pair of numbers. All of these numbers are derived from the tx_queue_enqueued and tx_queue_available_storage fields of TX_QUEUE.
Read	Gives the address of the next message that will be read with tx_queue_receive. This field corresponds to the tx_queue_read field of TX_QUEUE.
Write	Gives the address where the next message sent with tx_queue_send will be stored. This field corresponds to the tx_queue_write field of TX_QUEUE.
Start	Gives the address of the beginning of the message queue storage area. This field corresponds to the tx_queue_start field of TX_QUEUE.
Queue End	Gives the address of the end of the message queue storage area. This field corresponds to the tx_queue_end field of TX_QUEUE.

Control Block	Displays the address of the queue control block, which is a variable of type TX_QUEUE. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the queue control block. The **Data Explorer** window displays useful information that is not included in the **Queue Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended on an attempt to access the queue. This field corresponds to the tx_queue_suspended_count field of TX_QUEUE. If the queue is empty, the threads listed are suspended on calls to tx_queue_receive. If the queue is full, the threads listed are suspended on calls to tx_queue_send.
Suspended Threads List	Gives information about any threads currently suspended on an attempt to access the queue. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window for that thread (see "The Thread Information Window" on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create, this entry displays the address of its thread control block. • **Timeout** — Indicates the number of timer ticks before the thread will abort the attempted queue access with a return value of `TX_QUEUE_EMPTY` or `TX_QUEUE_FULL`. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to tx_queue_receive or tx_queue_send. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Queue List	Displays the **Queue List** window, which contains a list of all message queues in the system (see "The Queue List Window" on page 28).

Chapter 4

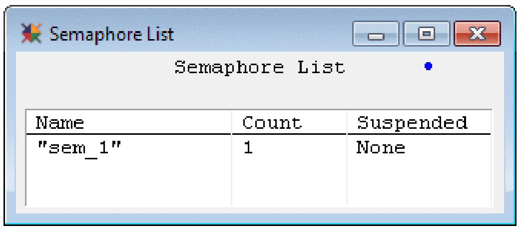
Semaphores

Contents

This chapter describes windows that display detailed information about the semaphores in your application.

The Semaphore List Window

The **Semaphore List** window shows a list of all semaphores in the system, arranged in the order in which they were created. To display this window, click the **Semaphores** button in the **ThreadX Information** window.



The list is generated by following a linked list, starting with the semaphore pointed to by the global variable `_tx_semaphore_created_ptr` and continuing with the tx_semaphore_created_next field of each semaphore control block TX_SEMAPHORE. A total of `_tx_semaphore_created_count` semaphores are shown.

When a semaphore is deleted, it disappears from this list.

You can double-click any semaphore in the **Semaphore List** window to display a **Semaphore Information** window (see "The Semaphore Information Window" on page 35).

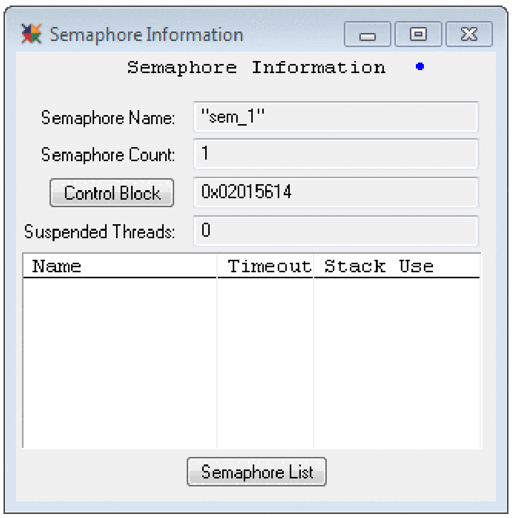
The **Semaphore List** has three columns: **Name**, **Count**, and **Suspended**. Each of these is described next.

Name	Displays the name of the semaphore, as given in the call to `tx_semaphore_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the semaphore control block TX_SEMAPHORE. This entry corresponds to the tx_semaphore_name field of TX_SEMAPHORE.
Count	Gives the count of the semaphore. Semaphore counts range from `0` to `0xffffffff`. This entry corresponds to the tx_semaphore_count field of TX_SEMAPHORE.

Suspended	Indicates the number of threads currently suspended on an attempt to get the semaphore with a call to `tx_semaphore_get`, or displays `None` if no threads are suspended. This entry corresponds to the tx_semaphore_suspended_count field of TX_SEMAPHORE.

The Semaphore Information Window

The **Semaphore Information** window shows detailed information about an individual semaphore.



To display this window, double-click a semaphore in the **Semaphore List** window, or view a variable of type TX_SEMAPHORE in the MULTI Debugger window. The information in the **Semaphore Information** window is derived from various fields within the semaphore control block TX_SEMAPHORE.

If a semaphore is deleted while a **Semaphore Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the semaphore control block.

Each field and button of the **Semaphore Information** window is described below.

Semaphore Name	Displays the name of the semaphore, as given in the call to `tx_semaphore_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_semaphore_name field of TX_SEMAPHORE.

Semaphore Count	Gives the count of the semaphore. Semaphore counts range from `0` to `0xffffffff`. This field corresponds to the tx_semaphore_count field of TX_SEMAPHORE.
Control Block	Displays the address of the semaphore control block, which is a variable of type TX_SEMAPHORE. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the semaphore control block. The **Data Explorer** window displays useful information that is not included in the **Semaphore Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended on an attempt to get the semaphore with a call to `tx_semaphore_get`. This field corresponds to the tx_semaphore_suspended_count field of TX_SEMAPHORE.
Suspended Threads List	Gives information regarding threads that are currently suspended on an attempt to get the semaphore. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window (see “The Thread Information Window” on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to `tx_thread_create`, this entry displays the address of the thread control block. • **Timeout** — Indicates the number of timer ticks before the thread will abort the attempted semaphore get. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to `tx_semaphore_get`. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Semaphore List	Displays the **Semaphore List** window, which contains a list of all semaphores in the system (see “The Semaphore List Window” on page 34).

Chapter 5

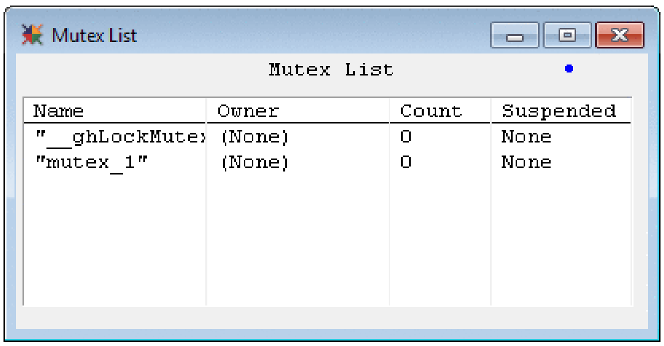
Mutexes

Contents

This chapter describes windows that display detailed information about the mutexes in your application.

The Mutex List Window

The **Mutex List** window shows a list of all mutexes in the system, arranged in the order in which they were created. To display this window, click the **Mutexes** button in the **ThreadX Information** window.



The **Mutex List** is generated by following a linked list, starting with the mutex pointed to by the global variable `_tx_mutex_created_ptr` and continuing with the tx_mutex_created_next field of each mutex control block TX_MUTEX. A total of `_tx_mutex_created_count` mutexes are shown.

When a mutex is deleted, it is removed from this list.

You can double-click any mutex in the **Mutex List** window to display a **Mutex Information** window (see "The Mutex Information Window" on page 39).

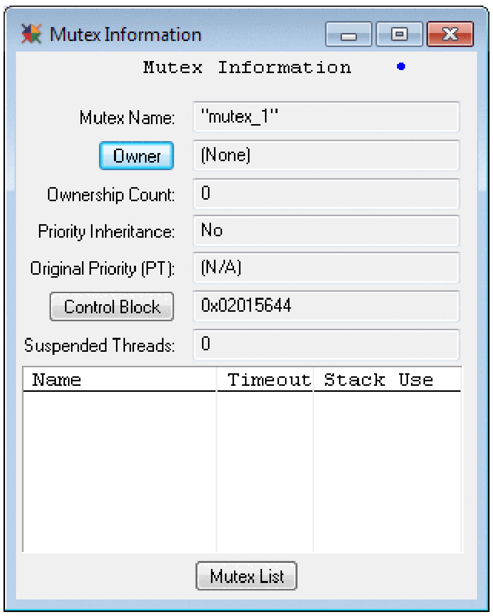
The **Mutex List** has four columns: **Name**, **Owner**, **Count**, and **Suspended**. Each of these is described next.

Name	Displays the name of the mutex, as given in the call to tx_mutex_create. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the mutex control block TX_MUTEX. This entry corresponds to the tx_mutex_name field of TX_MUTEX.
Owner	Gives the name of the thread that currently owns the mutex, or displays `(None)` if the ownership count is zero. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create when the owner thread was created, this entry displays the address of the thread control block. This entry is derived from the tx_mutex_owner field of TX_MUTEX.

Count	Gives the mutex ownership count. This entry corresponds to the tx_mutex_ownership_count field of TX_MUTEX.
Suspended	Indicates the number of threads currently suspended on attempts to get the mutex, or displays `None` if no threads are suspended. This field corresponds to the tx_mutex_suspended_count field of TX_MUTEX.

The Mutex Information Window

The **Mutex Information** window shows detailed information about an individual mutex.



To display this window, double-click a mutex in the **Mutex List** window, or view a variable of type TX_MUTEX in the MULTI Debugger window. The information in this window is derived from various fields within the mutex control block TX_MUTEX.

If a mutex is deleted while a **Mutex Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the mutex control block.

Each of the fields and buttons in the **Mutex Information** window is described below.

Mutex Name	Displays the name of the mutex, as given in the call to `tx_mutex_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_mutex_name field of TX_MUTEX.
Owner	Gives the name of the thread that currently owns the mutex, or displays `(None)` if the ownership count is zero. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create when the owner thread was created, this field displays the address of the thread control block. This field is derived from the tx_mutex_owner field of TX_MUTEX.
Ownership Count	Indicates the number of times the thread owner has called `tx_mutex_get` without a corresponding `tx_mutex_put`. If the ownership count is zero, no thread owns the mutex. This field corresponds to the tx_mutex_ownership_count field of TX_MUTEX.
Priority Inheritance	Indicates whether the mutex supports priority inheritance. This field corresponds to the tx_mutex_inherit field of TX_MUTEX.
Original Priority (PT)	Gives the original priority information (before any priority inheritance occurred) of the owner thread. Two numbers are displayed. The first number is the original priority level of the thread, which corresponds to the tx_mutex_original_priority field of TX_MUTEX. The second number, in parentheses, is the original preemption threshold of the thread and corresponds to the tx_mutex_original_threshold field of TX_MUTEX.
Control Block	Displays the address of the mutex control block, which is a variable of type TX_MUTEX. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the mutex control block. The **Data Explorer** window displays useful information that is not included in the **Mutex Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended on calls to `tx_mutex_get`. This field corresponds to the tx_mutex_suspended_count field of TX_MUTEX.

Suspended Threads List	Gives information about threads that are currently suspended on an attempt to acquire the mutex with `tx_mutex_get`. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window (see “The Thread Information Window” on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create, this entry displays the address of the thread control block. • **Timeout** — Indicates the number of timer ticks before the thread will abort the attempted mutex access with a return value of `TX_NOT_AVAILABLE`. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to tx_mutex_receive or tx_mutex_send. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Mutex List	Displays the **Mutex List** window, which contains a list of all mutexes in the system (see “The Mutex List Window” on page 38).

Chapter 6

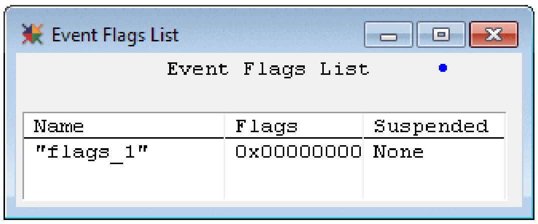
Event Flags Groups

Contents

This chapter describes windows that display detailed information about the event flags in your application.

The Event Flags List Window

The **Event Flags List** window shows a list of all event flags groups in the system, arranged in the order in which they were created. To display this window, click the **Event Flag Groups** button in the **ThreadX Information** window.



The **Event Flags List** is generated by following a linked list, starting with the event flags group pointed to by the global variable `_tx_event_flags_created_ptr` and continuing with the tx_event_flags_group_created_next field of each event flags group control block TX_EVENT_FLAGS_GROUP. A total of `_tx_event_flags_created_count` event flags groups are shown.

When an event flags group is deleted, it is removed from this list.

You can double-click any event flags group in the **Event Flags List** to display an **Event Flags Information** window (see “The Event Flags Information Window” on page 45).

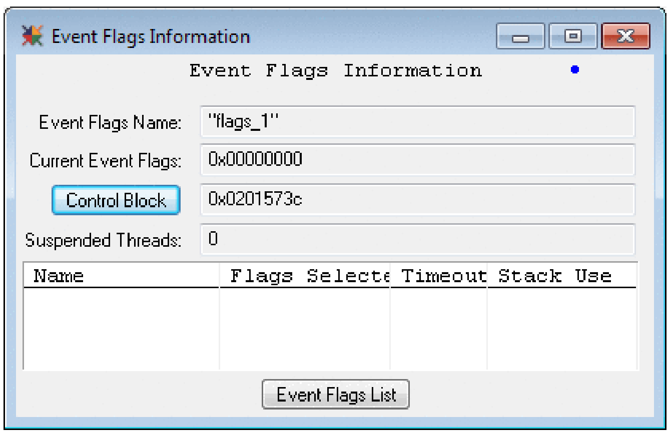
The **Event Flags List** has three columns: **Name**, **Flags**, and **Suspended**. Each of these is described next.

Name	Displays the name of the event flags group, as given in the call to `tx_event_flags_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the event flags group control block TX_EVENT_FLAGS_GROUP. This entry corresponds to the tx_event_flags_group_name field of TX_EVENT_FLAGS_GROUP.
Flags	Gives the status of the current event flags in hexadecimal format. Each event flags group contains 32 binary event flags. This entry corresponds to the tx_event_flags_group_current field of TX_EVENT_FLAGS_GROUP.

Suspended	Indicates the number of threads currently suspended while waiting for event flags to satisfy conditions specified in a call to `tx_event_flags_get`, or displays `None` if no threads are suspended. This entry corresponds to the tx_event_flags_group_suspended_count field of TX_EVENT_FLAGS_GROUP.

The Event Flags Information Window

The **Event Flags Information** window shows detailed information about an individual event flags group.



To display this window, double-click an event flags group in the **Event Flags List**, or view a variable of type TX_EVENT_FLAGS_GROUP in the MULTI Debugger window. The information in this window is derived from various fields within the event flags group control block TX_EVENT_FLAGS_GROUP.

If an event flags group is deleted while an **Event Flags Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the event flags group control block.

The information provided in the **Event Flags Information** window is described below.

Event Flags Name	Displays the name of the event flags group, as given in the call to `tx_event_flags_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_event_flags_group_name field of TX_EVENT_FLAGS_GROUP.

Current Event Flags	Indicates the status of the current event flags in hexadecimal format. Each event flags group contains 32 binary event flags. This field corresponds to the tx_event_flags_group_current field of TX_EVENT_FLAGS_GROUP.
Control Block	Displays the address of the event flags group control block, which is a variable of type TX_EVENT_FLAGS_GROUP. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the event flags group control block. The **Data Explorer** window displays useful information that is not included in the **Event Flags Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended while waiting for event flags to satisfy conditions specified in a call to `tx_event_flags_get`. This field corresponds to the tx_event_flags_group_suspended_count field of TX_EVENT_FLAGS_GROUP.

Suspended Threads List	Gives information about threads that are currently suspended while waiting for event flags to satisfy conditions specified in a call to `tx_event_flags_get`. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window (see “The Thread Information Window” on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create, this entry displays the address of the thread control block. • **Flags Selected** — Identifies the event flags that will satisfy the waiting thread's conditions. Two values are displayed. The first value, given in hexadecimal format, shows the flags that the thread requested. The second value contains one or two characters that show whether the thread is waiting for all or any of the event flags and whether the event flags will be cleared once the thread's requested event flags are satisfied. The first of these characters can be an `&` or `|` character, where `&` means that the thread is waiting for all of its requested event flags and `|` means that the thread will be satisfied by any one of its event flags being set. The second character is a `C` if the event flags specified by the thread will be cleared (set to zero) after they satisfy a thread's request. Otherwise, the second character is blank. • **Timeout** — Indicates the number of timer ticks before the thread will abort waiting for the event flags group to satisfy the thread's specified conditions. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to tx_event_flags_get. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Event Flags List	Displays the **Event Flags List** window, which contains a list of all event flags groups in the system (see “The Event Flags List Window” on page 44).

Chapter 7

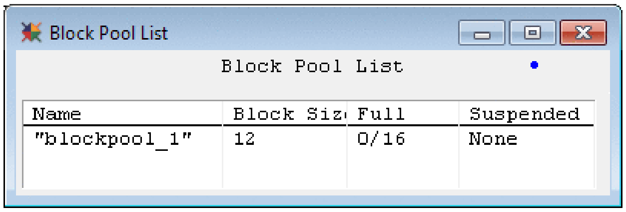
Memory Block Pools

Contents

This chapter describes windows that display detailed information about the memory block pools in your application.

The Block Pool List Window

The **Block Pool List** window shows a list of all memory block pools in the system, arranged in the order in which they were created. To display this window, click the **Block Pools** button in the **ThreadX Information** window.



The **Block Pools List** is generated by following a linked list, starting with the memory block pool pointed to by the global variable `_tx_block_pool_created_ptr` and continuing with the tx_block_pool_created_next field of each memory block pool control block TX_BLOCK_POOL. A total of `_tx_block_pool_created_count` pools are shown.

When a memory block pool is deleted, it is removed from this list.

You can double-click any memory block pool in the **Block Pool List** window to display a **Block Pool Information** window (see “The Block Pool Information Window” on page 52).

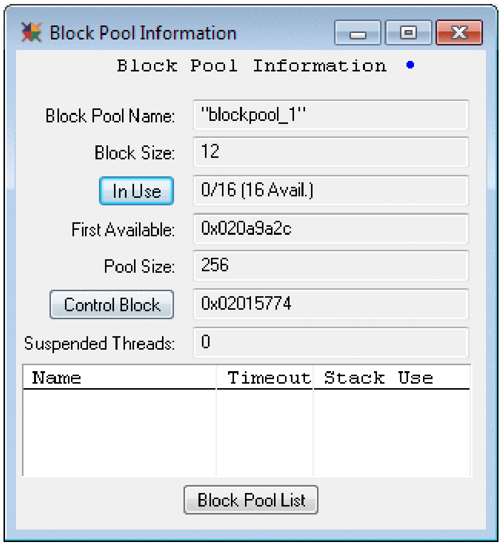
The **Block Pool List** has four columns: **Name**, **Block Size**, **Full**, and **Suspended**. Each of these is described next.

Name	Displays the name of the memory block pool, as given in the call to `tx_block_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the memory block pool control block TX_BLOCK_POOL. This entry corresponds to the tx_block_pool_name field of TX_BLOCK_POOL.

Block Size	Gives the size, in bytes, of each memory block in the pool. Block sizes displayed here are rounded up by the ThreadX kernel to an even multiple of 4 bytes in order to allow suitable alignment for the one pointer of overhead. This entry corresponds to the tx_block_pool_block_size field of TX_BLOCK_POOL.
Full	Indicates the number of memory blocks currently allocated. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of blocks currently allocated, and the second number indicates the total number of memory blocks. This entry is derived from the tx_block_pool_available and tx_block_pool_total fields of TX_BLOCK_POOL.
Suspended	Indicates the number of threads currently suspended on an attempt to allocate a block with a call to `tx_block_allocate`, or displays `None` if no threads are suspended. This entry corresponds to the tx_block_pool_suspended_count field of TX_BLOCK_POOL.

The Block Pool Information Window

The **Block Pool Information** window shows detailed information about an individual memory block pool.



To display this window, double-click a memory block pool in the **Block Pool List** window, or view a variable of type TX_BLOCK_POOL in the MULTI Debugger window. The information in the **Block Pool Information** window is derived from various fields within the memory block pool control block TX_BLOCK_POOL.

If a memory block pool is deleted while a **Block Pool Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the memory block pool control block.

To view detailed information about the memory blocks in a pool, click the **In Use** button to display the **Block Pool Contents** window (see "The Block Pool Contents Window" on page 55).

The fields and buttons in the **Block Pool Information** window are described next.

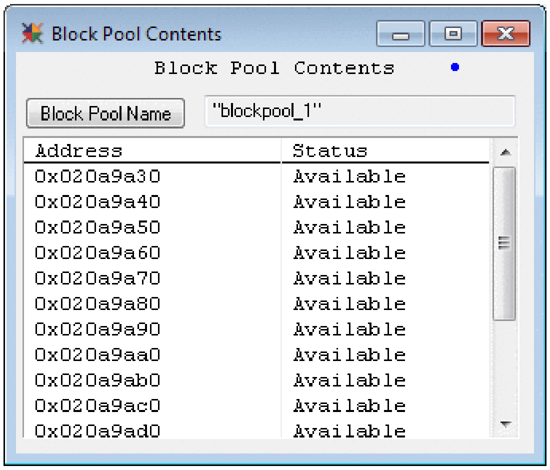
Block Pool Name	Displays the name of the memory block pool, as given in the call to `tx_block_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_block_pool_name field of TX_BLOCK_POOL.

Block Size	Gives the size, in bytes, of each memory block in the pool. Block sizes displayed here are rounded up by the ThreadX kernel to an even multiple of 4 bytes to allow suitable alignment for the one pointer of overhead. This field corresponds to the tx_block_pool_block_size field of TX_BLOCK_POOL.
In Use	Shows the number of memory blocks currently allocated. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of blocks currently allocated, and the second number indicates the total number of memory blocks. The number of available blocks (the total number of memory blocks minus the number of blocks allocated) is also shown after this pair of numbers. All of these numbers are derived from the tx_block_pool_available and tx_block_pool_total fields of TX_BLOCK_POOL. Clicking the **In Use** button displays a **Block Pool Contents** window that shows which specific blocks are allocated (see “The Block Pool Contents Window” on page 55).
First Available	Points to the first available memory block, or zero if the block pool is completely allocated. The address displayed is actually 4 bytes before the memory block that will be allocated upon a call to `tx_block_allocate`. These 4 bytes of overhead contain a pointer. Available memory blocks are kept in a singly-linked list starting with the first available block. In allocated blocks, the pointer points to the memory block pool control block, which allows blocks to be released on a call to `tx_block_release` without specifying the block pool from which the block was allocated. This field corresponds to the tx_block_pool_available_list field of TX_BLOCK_POOL.
Pool Size	Shows the number of bytes in the memory block pool storage area. This field corresponds to the tx_block_pool_size field of TX_BLOCK_POOL.
Control Block	Gives the address of the memory block pool control block, which is a variable of type TX_BLOCK_POOL. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the memory block pool control block. The **Data Explorer** window displays useful information that is not included in the **Block Pool Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended on an attempt to allocate a block from the memory pool with a call to `tx_block_allocate`. This field corresponds to the tx_block_pool_suspended_count field of TX_BLOCK_POOL.

Suspended Threads List	Gives information about threads that are currently suspended on an attempt to allocate a block from the memory pool. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window (see "The Thread Information Window" on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create, this entry displays the address of the thread control block. • **Timeout** — Indicates the number of timer ticks before the thread will abort the `tx_block_allocate` call with a return value of `TX_NO_MEMORY`. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to tx_block_allocate. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Block Pool List	Displays the **Block Pool List** window, which contains a list of all memory block pools in the system (see "The Block Pool List Window" on page 50).

The Block Pool Contents Window

The **Block Pool Contents** window shows a list of all blocks in a memory block pool. To display this window, click the **In Use** button in the **Block Pool Information** window.



The information in the **Block Pool Contents** window is derived from various fields within the memory block pool control block TX_BLOCK_POOL.

You can double-click any memory block listed in the **Block Pool Contents** window to view the contents of memory at that location.

The fields and buttons of the **Block Pool Contents** window are described below.

Block Pool Name	Displays the name of the memory block pool, as given in the call to `tx_block_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the memory block pool control block TX_BLOCK_POOL. This entry corresponds to the tx_block_pool_name field of TX_BLOCK_POOL. Clicking the **Block Pool Name** button displays the **Block Pool Information** window for the memory block pool (see "The Block Pool Information Window" on page 52).

Memory Block List	Gives information about the memory blocks in the block pool. Each column in the list is described below. Double-clicking any listed memory block displays the contents of memory at that location. • **Address** — Gives the address in memory where the block resides. The 4 bytes preceding this address contain a pointer. If this pointer points to the memory block pool control block, the block is in use. Otherwise, the block is available. • **Status** — Displays either `In Use` or `Available` depending on the value of the pointer preceding the memory block, as described above.

Chapter 8

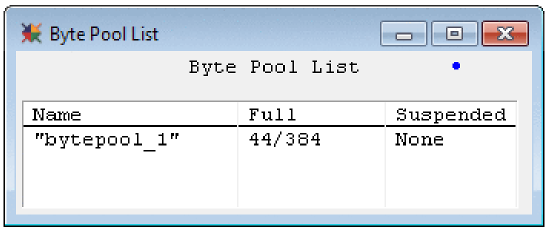
Memory Byte Pools

Contents

This chapter describes windows that display detailed information about the memory byte pools in your application.

The Byte Pool List Window

The **Byte Pool List** window shows a list of all memory byte pools in the system, arranged in the order in which they were created. To display this window, click the **Byte Pools** button in the **ThreadX Information** window.



The list is generated by following a linked list, starting with the memory byte pool pointed to by the global variable `_tx_byte_pool_created_ptr` and continuing with the _tx_byte_pool_created_next field of each memory byte pool control block TX_BYTE_POOL. A total of `_tx_byte_pool_created_count` pools are shown.

When a memory byte pool is deleted, it is removed from this list.

You can double-click any memory byte pool in the **Byte Pool List** to display a **Byte Pool Information** window (see "The Byte Pool Information Window" on page 59).

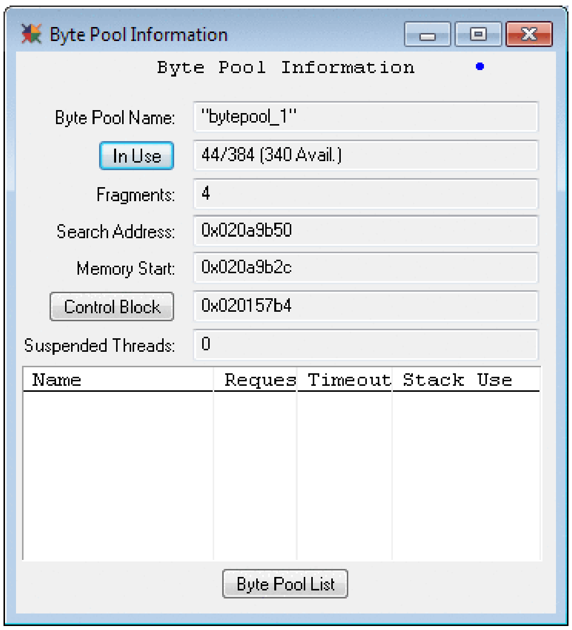
The **Byte Pool List** has three columns: **Name**, **Full**, and **Suspended**. Each of these is described next.

Name	Displays the name of the memory byte pool, as given in the call to `tx_byte_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the memory byte pool control block TX_BYTE_POOL. This entry corresponds to the tx_byte_pool_name field of TX_BYTE_POOL.
Full	Shows the number of bytes currently allocated from the pool. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of bytes currently allocated, and the second number indicates the total number of bytes. These numbers are derived from the tx_byte_pool_available and tx_byte_pool_size fields of TX_BYTE_POOL.

Suspended	Shows the number of threads currently suspended on an attempt to allocate memory from the pool with a call to `tx_byte_allocate`, or displays `None` if no threads are suspended. This entry corresponds to the tx_byte_pool_suspended_count field of TX_BYTE_POOL.

The Byte Pool Information Window

The **Byte Pool Information** window shows detailed information about an individual memory byte pool.



To display this window, double-click a byte pool in the **Byte Pool List**, or view a variable of type TX_BYTE_POOL in the MULTI Debugger window. The information in this window is derived from various fields within the memory byte pool control block TX_BYTE_POOL.

If a memory byte pool is deleted while a **Byte Pool Information** window for it exists, the window does not automatically disappear; the window continues to show the contents of the memory byte pool control block.

To view detailed information about the fragments in a byte pool, click the **In Use** button to display the **Byte Pool Contents** window (see “The Byte Pool Contents Window” on page 62).

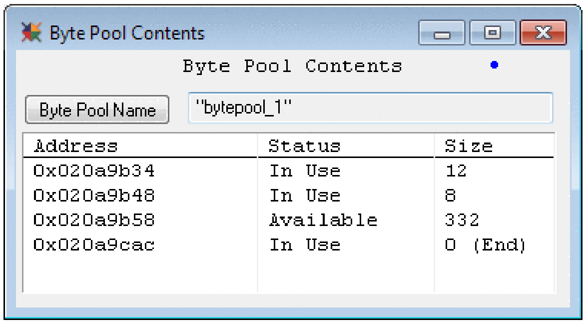
The fields and buttons of the **Byte Pool Information** window are described in the table below.

Byte Pool Name	Gives the name of the memory byte pool, as given in the call to `tx_byte_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_byte_pool_name field of TX_BYTE_POOL.
In Use	Shows the number of bytes currently allocated. Two numbers separated by a forward slash (`/`) are displayed. The first number indicates the number of bytes currently allocated, and the second number indicates the total number of bytes. The number of available bytes (the total number of bytes minus the number of bytes allocated) is shown after this pair of numbers. Note that the available byte count does not compensate for the two pointers of overhead that each memory fragment requires. The numbers in this field are derived from the tx_byte_pool_available and tx_byte_pool_size fields of TX_BYTE_POOL. Clicking the **In Use** button displays a **Byte Pool Contents** window that shows all allocated and unallocated fragments in the byte pool (see "The Byte Pool Information Window" on page 59).
Fragments	Indicates the number of fragments in the memory byte pool. This value is derived from the tx_byte_pool_fragments field of TX_BYTE_POOL.
Search Address	Points to the first unallocated memory fragment that will be searched during tx_byte_allocate. The address is set to the last fragment that was released. This field corresponds to the tx_byte_pool_search field of TX_BYTE_POOL.
Memory Start	Points to the start of the byte pool and corresponds to the tx_byte_pool_start field of TX_BYTE_POOL.
Control Block	Gives the address of the memory byte pool control block, which is a variable of type TX_BYTE_POOL. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the memory byte pool control block. The **Data Explorer** window displays useful information that is not included in the **Byte Pool Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Suspended Threads	Indicates the number of threads currently suspended on an attempt to allocate memory from the memory pool with a call to tx_byte_allocate. This field corresponds to the tx_byte_pool_suspended_count field of TX_BYTE_POOL.

Suspended Threads List	Gives information about threads that are currently suspended on an attempt to allocate memory from the memory pool. Each column in the list is described below. Double-click any listed thread to display a **Thread Information** window (see "The Thread Information Window" on page 19). • **Name** — Gives the name of the thread. If a `0` (null pointer) was passed as the `name_ptr` argument to tx_thread_create, this entry displays the address of its thread control block. • **Request** — Indicates the number of bytes requested in the memory_size argument to tx_byte_allocate. Sometimes a request causes a thread to suspend even though enough memory appears to be available. This can occur if the two pointers necessary in each memory fragment have not been allowed for or if the pool is too fragmented to satisfy the request. • **Timeout** — Indicates the number of timer ticks before the thread will abort the `tx_byte_allocate` call with a return value of `TX_NO_MEMORY`. This entry shows `Forever` if `TX_WAIT_FOREVER` was passed as the **wait_option** to tx_byte_allocate. • **Stack Use** — Shows the amount of stack currently in use by the thread.
Byte Pool List	Displays the **Byte Pool List** window, which contains a list of all memory byte pools in the system (see "The Byte Pool List Window" on page 58).

The Byte Pool Contents Window

The **Byte Pool Contents** window shows a list of all fragments in a memory byte pool.



To display this window, click the **In Use** button in the **Byte Pool Information** window. The information in the **Byte Pool Contents** window is derived from various fields within the memory byte pool control block TX_BYTE_POOL.

You can double-click any fragment listed in the **Byte Pool Contents** window to view the contents of memory at that location.

The information listed in the **Byte Pool Contents** window is described next.

Byte Pool Name	Displays the name of the memory byte pool, as given in the call to `tx_byte_pool_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the memory byte pool control block TX_BYTE_POOL. This entry corresponds to the tx_byte_pool_name field of TX_BYTE_POOL. Clicking the **Byte Pool Name** button displays the **Byte Pool Information** window for the memory byte pool (see "The Byte Pool Information Window" on page 59).

Memory Byte List	Gives information about the fragments in the byte pool. Each column in this list is described below. Double-clicking any listed fragment displays the contents of memory at that location. • **Address** — Gives the address in memory of the fragment. The location 8 bytes before this address contains a pointer. If this pointer points to the memory byte pool control block, the fragment is in use. Otherwise, the fragment is available. • **Status** — Shows either `In Use` or `Available` depending on whether the location 4 bytes before the fragment contains the value `TX_BYTE_BLOCK_FREE` (`0xffffeeee`). • **Size** — Indicates the size of the byte pool fragment in bytes. All byte pools end with a zero-byte fragment that is shown as `0` (End).

Chapter 9

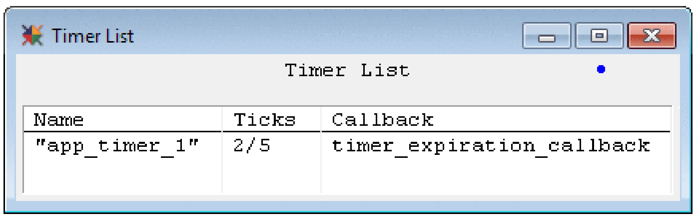
Application Timers

Contents

This chapter describes windows that display detailed information about the application timers in your application.

The Timer List Window

The **Timer List** window shows a list of all application timers in the system, arranged in the order in which they were created. This list can be displayed by clicking the **Application Timers** button in the **ThreadX Information** window.



The **Timer List** is generated by following a linked list, starting with the timer pointed to by the global variable `_tx_timer_created_ptr` and continuing with the tx_timer_created_next field of each timer control block TX_TIMER. A total of `_tx_timer_created_count` timers are shown.

When a timer is deleted, it disappears from this list.

You can double-click any timer in the **Timer List** to display a **Timer Information** window (see “The Timer Information Window” on page 68).

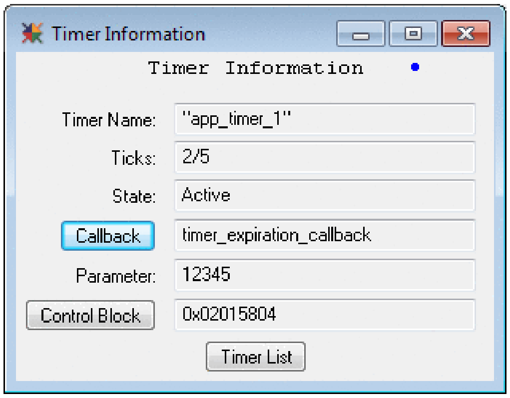
The **Timer List** has three columns: **Name**, **Ticks**, and **Callback**. Each of these is described next.

Name	Displays the name of the timer, as given in the call to `tx_timer_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this entry displays the address of the timer control block TX_TIMER. This entry corresponds to the tx_timer_name field of TX_TIMER.

Ticks	Indicates the number of initial timer ticks and the timer reschedule tick value if the timer is active. If the timer is inactive, this field displays `Inactive`. For active timers, two numbers separated by a forward slash (`/`) are displayed. The first number specifies the current number of initial ticks, and the second number specifies the number of ticks with which the timer will be rescheduled after it expires. Both numbers range from `0` to `0xffffffff`. A zero value for the second number specifies a one-shot timer. The current tick value corresponds to the tx_timer_internal_remaining_ticks field of the tx_timer_internal structure within TX_TIMER. The reschedule tick value corresponds to the tx_timer_internal_re_initialize_ticks of the tx_timer_internal structure within TX_TIMER.
Callback	Gives the name of the function called when the timer expires. If no debugging information is available, this entry may be displayed as an offset from a known label or as an address in hexadecimal format. This entry corresponds to the tx_timer_internal_timeout_function of the tx_timer_internal structure within TX_TIMER.

The Timer Information Window

The **Timer Information** window shows detailed information about an individual timer.



To display this window, double-click a timer in the **Timer List** window, or view a variable of type TX_TIMER in the MULTI Debugger window. The information in this window is derived from various fields within the timer control block TX_TIMER.

Each of the fields and buttons in the **Timer Information** window is described below.

Timer Name	Displays the name of the timer, as given in the call to `tx_timer_create`. If a `0` (null pointer) was passed as the `name_ptr` argument, this field displays `(None)`. This field corresponds to the tx_timer_name field of TX_TIMER.
Ticks	Indicates the number of initial timer ticks and the timer reschedule tick value. Two numbers are listed, separated by a forward slash (`/`). The first number specifies the initial number of ticks when the timer is created, and the second number specifies the number of ticks for all timer expirations after the first. Both numbers range from `0` to `0xffffffff`. A zero value for the second number specifies a one-shot timer. The initial tick value corresponds to the tx_timer_internal_remaining_ticks field of the tx_timer_internal structure within TX_TIMER. The reschedule tick value corresponds to the tx_timer_internal_re_initialize_ticks of the tx_timer_internal structure within TX_TIMER.

State	Shows the current state of the timer (`Active` or `Inactive`). This value is derived from the tx_timer_internal_list_head field of the tx_timer_internal structure within TX_TIMER. The timer is active only if tx_timer_internal_list_head is non-zero.
Callback	Gives the name of the function called when the timer expires. If no debugging information is available, then this field may be displayed as an offset from a known label or as an address in hexadecimal format. This field corresponds to the tx_timer_internal_timeout_function of the tx_timer_internal structure within TX_TIMER. Clicking the **Callback** button shows the callback function.
Parameter	Indicates the parameter passed to the callback function when the timer expires. This field corresponds to the tx_timer_internal_timeout_param field of the tx_timer_internal structure within TX_TIMER.
Control Block	Shows the address of the timer control block, which is a variable of type TX_TIMER. See the ThreadX header file **tx_api.h** for the definition of this type. Clicking the **Control Block** button opens a **Data Explorer** window on the timer control block. The **Data Explorer** window displays useful information that is not included in the **Timer Information** window (for more information, see the documentation about the Data Explorer in the *MULTI: Debugging* book).
Timer List	Clicking this button displays the **Timer List** window, which contains a list of all timers in the system (see "The Timer List Window" on page 66).

Part II

Using the MULTI EventAnalyzer for ThreadX

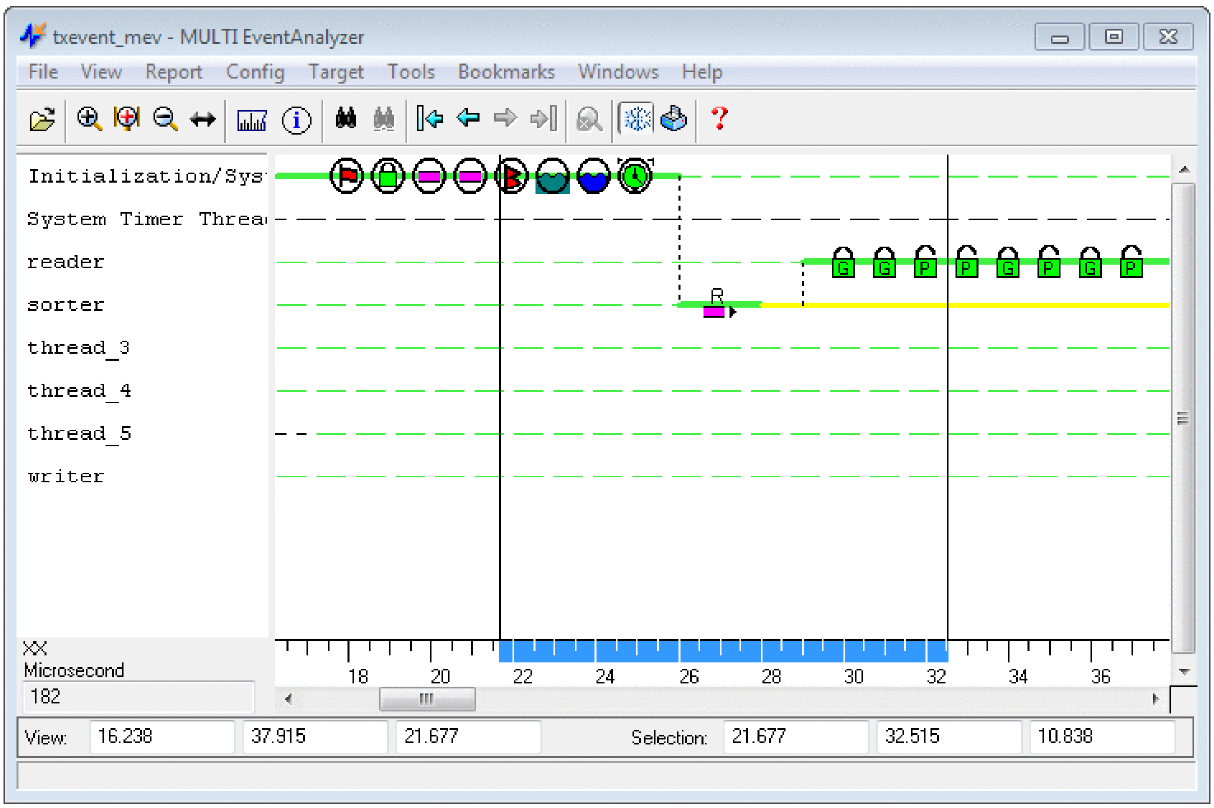
Chapter 10

Introduction to the MULTI EventAnalyzer for ThreadX

Contents

The MULTI EventAnalyzer helps developers understand the dynamic behavior of a target that uses the ThreadX real-time kernel by providing a graphical representation of system activities as they occur over time. The EventAnalyzer complements the MULTI Debugger, which displays detailed system information at a single point in time.

The event logging feature available with the ThreadX kernel allows the system to record specific event information such as ThreadX service calls, context switches, interrupts, and user-defined events as they occur. This event data is transferred to the host system and is viewed and analyzed with the MULTI EventAnalyzer.



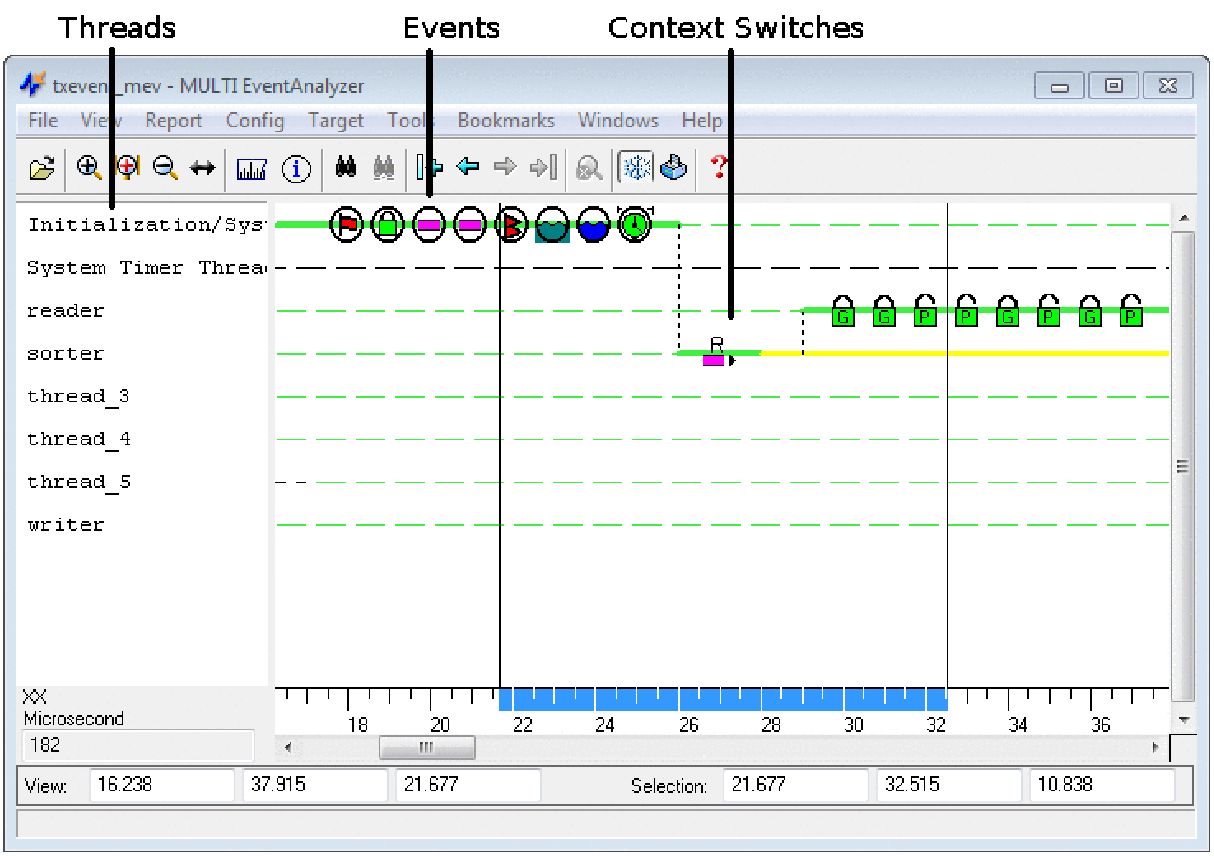
The EventAnalyzer displays details about the status of each thread and about events related to that thread, and includes a variety of controls that enable you to view the event data.

Basic Operation

When event logging is enabled, the ThreadX kernel logs events and status changes to a target-resident buffer as they occur. At any point, this memory region can be retrieved from the target and saved to a data file on the host. The EventAnalyzer can then display the data graphically.

The EventAnalyzer reads the data file as a series of events and states. The various threads on the target system can change execution state as the target runs. System events occur depending on the behavior of the program and of the system.

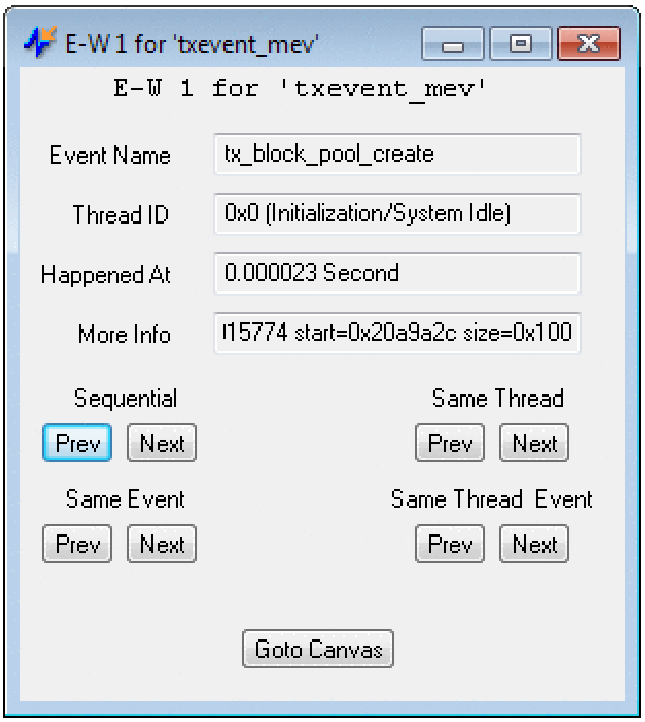
The main display of the EventAnalyzer (pictured below) provides a graphical depiction of events, context switches, and status changes as they occur over time.



To view additional details about an event, click any object in the view graph canvas.

Every event displayed by the EventAnalyzer has an "object view" that is displayed by double-clicking the event icon in the EventAnalyzer canvas or by using the right-click menu choice **Show Object**. The object view displays the extra data along with other pertinent, event-specific information.

An example of the object view for an event is shown below. Note the extra data displayed in the **More Info** field.



The standard ThreadX system events are classified as follows:

- **Thread context switches** — A *context switch* refers to the moment when the kernel changes the current thread running on the CPU. This can occur when one thread preempts another, when the running thread suspends itself, or when the running thread suspends on a resource. In the EventAnalyzer canvas, a vertical dashed line represents a thread context switch, and horizontal lines with differing line styles indicate the status of each thread.
- **Exceptions and interrupts** — An *exception* is an event that causes the processor to suspend its current operation immediately and perform some processing to service the exception. Exceptions caused by external devices are called hardware interrupts. These can occur asynchronously with respect to the execution of code. Other exceptions may be caused by the synchronous execution of code. Some examples of synchronous exceptions are division by

zero, memory protection violations, illegal instructions, and unaligned access exceptions.

- **Service calls** — A *service call* is an event that occurs when a thread calls a ThreadX service function, thus causing the kernel to perform an operation on behalf of the thread. Common examples include operations on semaphores, sending and receiving messages, and thread manipulation. Because ThreadX service calls are the interface to the kernel, logging and analysis of these events is usually the most important function of the EventAnalyzer.
- **User events** — You can insert code into your application to log events and record specific system data, such as the values of particular variables or expressions related to those events. For more information, see "User-Defined Events" on page 82.

Between the time it is created and destroyed, a thread will be in one of the following states at any given moment:

- `Ready` — The thread is ready for execution, but the ThreadX scheduler is currently running a higher-priority thread or another same-priority thread. A typical system might contain several ready threads; however, the scheduler executes only one thread at a time, based on priority and the order in which they became ready.
- `Executing` — The thread is currently executing on the CPU.
- `Suspended` — The thread is not ready for execution.
- `Completed` — The thread has returned from its entry function.
- `Terminated` — The thread has been terminated (either by itself or another thread) by a call to `tx_thread_terminate`.

When the status of a thread changes (for example, a `Ready` thread is set `Executing` by the scheduler), this information is logged as a thread status change event.

The Effect of Event Logging on Run-Time Performance

Event logging requires a small amount of system overhead, which is directly proportional to the number of events logged.

This section discusses factors that may affect the level of intrusion into the target system. For more information, see Chapter 11, “Collecting Event Logging Data” on page 79.

Basic Logging Instrumentation

With event logging support present and disabled, the effect on run-time performance is minimal. When the preprocessor symbol `TX_ENABLE_EVENT_LOGGING` is set while building the kernel, ThreadX includes the kernel instrumentation necessary for event logging. Therefore, even when event logging is disabled and no data is being logged, a small amount of overhead exists due to run-time checks by the system to determine if logging is enabled.

The ThreadX library can be built without logging support, which removes all event logging overhead.

Quantity of Event Types

The optional event logging filter allows you to include or exclude certain types of events. This allows you to log only the events necessary for meaningful analysis. Logging more events uses up more of the available target memory.

Chapter 11

Collecting Event Logging Data

Contents

This chapter discusses issues relating to configuring your program for event logging.

Control and Filtering of Event Logging

Event logging is controlled at compile-time via conditional compilation. To implement event logging, use the following defines when compiling ThreadX or application source:

- `TX_ENABLE_EVENT_LOGGING` — (Main option) Enables event logging for any or all of the ThreadX source code. If this option is used anywhere, the **tx_initialize_high_level.c** file must be compiled with it as well.
- `TX_NO_EVENT_INFO` — (Sub-option) Suppresses the collection of "extra data" that ThreadX collects for each event. Each ThreadX event returns a predetermined set of information about the event, including the Event Name, the thread ID, the time at which the event occurred, and, in many cases, some extra data related to the event. Defining this symbol suppresses the collection of extra data.
- `TX_ENABLE_EVENT_FILTERS` — (Sub-option) Enables event filters, allowing you to control the types of events that are logged.

Note

By default, ThreadX's event logging code supports 16 thread names in applications. If you need to track more than 16 threads, change the value of the `TX_EL_TNIS` macro, which is defined in the **tx_el.h** include file. Then rebuild the ThreadX library.

ThreadX provides three routines that can be used to control event logging at run-time from within the application software. These routines require that event filtering be enabled as described above.

- `void _tx_el_event_log_on(void);` Instructs ThreadX to begin logging events, by clearing the internal event logging filter.
- `void _tx_el_event_log_off(void);` Instructs ThreadX to stop logging events, by setting all the event flag logging filters.
- `void _tx_el_event_filter_set(UINT filter);` Sets the logging event filter, which specifies the types of events to be excluded from logging. You will typically want to view only certain types of events. For example, in some

systems, interrupts and status events can occur frequently. Logging all of these events could return more data than can be reasonably analyzed or overflow the memory buffer. (The capacity of the target memory limits the quantity of data that can be retained.) In these cases, suppressing certain event types will allow enough memory for the desired events.

The event types that can be filtered and their bit mask values (which can be combined through a bitwise `OR` to filter out more events) are as follows:

- `TX_EL_FILTER_STATUS_CHANGE` (`0x0001`) — Events describing thread status changes such as when a thread changes from suspended to ready.
- `TX_EL_FILTER_INTERRUPTS` (`0x0002`) — Interrupt or exception events. In some systems, interrupts can occur at a very high rate, causing a flood of event data that may make the other data more difficult to visualize or may generate more events than the logging mechanism can handle. It might also be the case that these events should be removed in the actual ThreadX source code modules - typically the interrupt logic is isolated to a few assembly files in most versions of ThreadX.
- `TX_EL_FILTER_THREAD_CALLS` (`0x0004`) — Events associated with thread services in ThreadX (e.g., `tx_thread_resume`, `tx_thread_suspend`, etc.).
- `TX_EL_FILTER_TIMER_CALLS` (`0x0008`) — Events associated with application timer services in ThreadX (e.g., `tx_timer_activate`, `tx_timer_deactivate`, etc.).
- `TX_EL_FILTER_EVENT_FLAG_CALLS` (`0x0010`) — Events associated with event flag services in ThreadX (e.g., `tx_event_flags_get`, `tx_event_flags_set`, etc.).
- `TX_EL_FILTER_SEMAPHORE_CALLS` (`0x0020`) — Events associated with semaphore services in ThreadX (e.g., `tx_semaphore_get`, `tx_semaphore_put`, etc.).
- `TX_EL_FILTER_QUEUE_CALLS` (`0x0040`) — Events associated with queue services in ThreadX (e.g., `tx_queue_send`, `tx_queue_receive`, etc.).
- `TX_EL_FILTER_BLOCK_CALLS` (`0x0080`) — Events associated with memory block pool services in ThreadX (e.g., `tx_block_allocate`, `tx_block_release`, etc.).

- `TX_EL_FILTER_BYTE_CALLS` (`0x0100`) — Events associated with memory byte pool services in ThreadX (e.g., `tx_byte_allocate`, `tx_byte_release`, etc.).
- `TX_EL_FILTER_MUTEX_CALLS` (`0x0200`) — Events associated with mutex services in ThreadX (e.g., `tx_mutex_get`, `tx_mutex_prioritize`, etc.).
- `TX_EL_FILTER_ALL_EVENTS` (`0xFFFF`) — Disables collection of all events.
- `TX_EL_ENABLE_ALL_EVENTS` (`0x0000`) — [default] Enables collection of all events.

Note

Filtering events while the application runs is distinct from changing the visibility attribute of an event in the EventAnalyzer application. The event filter actually prevents the system from logging particular events; consequently, those events are not written to the data file. The visibility attribute merely *turns off* the selected event in the EventAnalyzer display canvas, but the event is logged, takes up memory, and will be present in the data file.

User-Defined Events

User-defined events allow the target to log events that are specific to your application. You can use user-defined events to enhance event logging capabilities. For example, if you need to determine when a particular piece of code executes, a user event can be logged in the code at that point. Such modifications can be useful in order to better understand how the target system is operating.

To implement a user-defined event you must:

- Modify the Application
- Modify the Configuration File

Modify the Application

The API service for logging a user-defined event in ThreadX is as follows:

```
VOID _tx_el_user_event_insert(UINT sub_type, ULONG info_1,
          ULONG info_2, ULONG info_3, ULONG info_4);
```

The parameters supplied to this service are defined by the application. ThreadX simply places this information along with the current thread pointer and time-stamp into the next entry in the event log.

For example, to track the time it takes to process an application buffer, you might insert code as follows:

```
/*Buffer is received, record a user-defined event.
 * Note that BUFFER_RECEIVED and my_buffer_ptr are defined
 * outside of this scope. */
_tx_el_user_event_insert(BUFFER_RECEIVED,
          (ULONG) my_buffer_ptr);
/* ... a bunch of processing */
/* Buffer is processed, record a user-defined event. Note
 * that BUFFER_PROCESSED is defined outside of this scope.*/
_tx_el_user_event_insert(BUFFER_PROCESSED,
          (ULONG) my_buffer_ptr);
```

Modify the Configuration File

For an event to appear in the EventAnalyzer, it must be defined in the ThreadX configuration file (**threadx.mc**).

The ThreadX configuration file already includes definitions for standard ThreadX events. User-defined events can be added by using the following syntax:

```
MEV_Event:4:0:MyUserEvent:userevent:MEV_Extra="count=%4D":MEV_Visible
```

This configuration file entry tells how to display a user-defined event, called `MyUserEvent` in this example. The **MEV_Event:4** field entries indicate that this entry describes a user-defined event. The **subtype** field, which in this example is 0, corresponds to the `sub_type` argument passed to the `_tx_el_user_event_insert` service. The **userevent** field entry indicate that the standard user-defined event icon should be used in the EventAnalyzer display. The display of any extra data is indicated by the `count=%4D` string.

Other kinds of user-defined events with different extra data formats can be defined by adding a new event with a different subtype to the **threadx.mc** file.

For more information about the configuration files, see Chapter 13, “EventAnalyzer Configuration Files” on page 107 To create *extra data* for events, see “Specifying Extra Data” on page 110.

Note

You must restart the EventAnalyzer for changes to your configuration file to take effect.

Retrieving Event Logging Data from the Target

See “Launching the EventAnalyzer” on page 88 for the typical way of retrieving and viewing event log data.

To perform the event log data retrieval, postprocessing, and EventAnalyzer actions separately, follow these steps:

1. Establish a debugging connection capable of reading the target memory.
2. Use the MULTI **memdump** command to retrieve the contents of the event log and write it to a file on the host computer system. For more information about this and other commands, see the *MULTI: Debugging Command Reference* book.

For example, to dump the ThreadX event log into a file called **my_events** on the host, enter the following:

```
memdump raw my_event_dump __ghsbegin_eventlog \
        (__ghsend_eventlog-__ghsbegin_eventlog)
```

The above operation places the contents of the event log into the file, **my_event_dump**, located on the host. If the Debugger generates an `unknown symbol` error (because the Debugger cannot find either of the special `__ghsbegin` symbols), examine the application map file or use the MULTI **map** command to determine the location and size of the `.eventlog` section. Then use explicit address and size arguments in the **memdump** command.

Once the event log has been placed into a host file, you must convert it into a format that is compatible with the EventAnalyzer. Do this by using the **txundump** utility provided with your distribution. The following shell command converts the raw event log into the proper format for the EventAnalyzer:

```
txundump my_event_dump my_events
```

The resulting **my_events.mes** file is ready to view with the EventAnalyzer.

Then use the Debugger **mev** command to launch the EventAnalyzer:

```
mev my_events
```

Modifying the Target Event Log Location

By default, ThreadX uses a statically-allocated buffer to store the event log. The size of the target buffer limits the number of events that can be acquired during an event logging session. If the buffer becomes full, the oldest events are overwritten with new events.

The memory buffer used to hold the event log data is found in the special program section, `.eventlog`. The size of this section determines the size of the event logging buffer and can be configured by changing the linker file.

You can store the event log in a more permanent location (for example off-board memory), by modifying the ThreadX code in **tx_el.c**.

Chapter 12

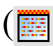
Viewing Event Data

Contents

Once you have collected event data for your application in a data file, you can view a graphical representation of the events in the EventAnalyzer. This chapter describes how to use the features of the EventAnalyzer to navigate the event log and select and view event data.

Launching the EventAnalyzer

The simplest way to retrieve and view event logging data is to use one of the following two methods:

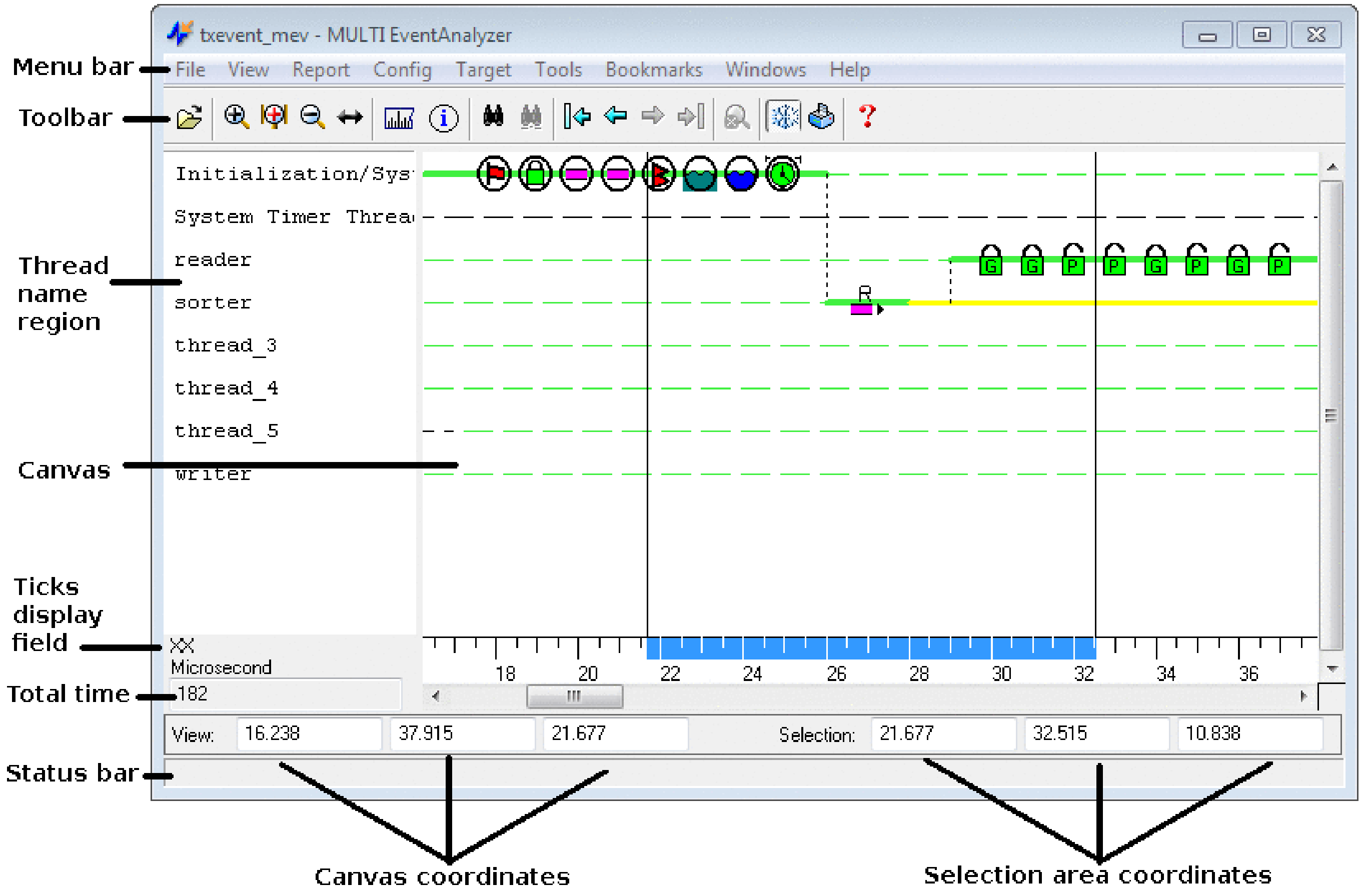
- From the Debugger choose **Tools → MULTI EventAnalyzer**
- From the Debugger, click the EventAnalyzer button ()

These will automatically dump any event log data from the `.eventlog` section to a file, postprocess that file, and launch the EventAnalyzer.

If you need to perform the event log data retrieval, postprocessing, and EventAnalyzer steps separately, see “Retrieving Event Logging Data from the Target” on page 84.

The EventAnalyzer Window

When you launch the EventAnalyzer, the following window appears:



The EventAnalyzer window contains the following components:

- *Thread name region* — Lists each thread for which events were logged during the event logging session. The first thread to appear in the data file appears at the top of the list and the last at the bottom.
- *Canvas* — Displays a line graph covering a time range that contains the following items:
    - *Threads* — Represented as horizontal line segments with different colors and line styles. For example, a thread that is `executing` may be represented by a thick green line. A change in the line color or style indicates a change in status.

  - *Events* — Represented by icons along the horizontal line segments.
  - *Context switches* — Represented by vertical line segments between threads.

  The canvas reads from left (earliest event) to right (latest event). The colors and styles of the line segments and the icons can be customized. You can zoom into the canvas to display events in greater detail or zoom out to display events over a greater time range. For more information about customizing the interface, see "Using the Legend" on page 96.

- *Ticks display field* — Indicates the elapsed time. To the left of the scale, the selected time unit of measure is displayed along with the scale reference time. The scale reference time provides the first several numerals of the scale value and the remaining numerals are found on the scale itself. For example, if the ticks display field reads `12.34XX Seconds` and a point on the time scale is 55, the elapsed time at that point in the canvas is 12.3455 seconds. By displaying numbers in this format, a greater number of axis labels can be displayed in the same space.
- *Canvas coordinate fields* — Indicate the exact time range currently visible in the canvas. Reading from left to right, the view fields display the starting time, the ending time, and the total time span visible. The time unit of measure is the same as that displayed in the ticks display field.

  The starting and ending times refer only to the time currently visible in the canvas and not necessarily the starting or ending time of the entire data file.

  View coordinates may also be entered into any of the view fields. After entering new coordinates, press **Enter** to update the canvas. When a new coordinate is entered into a view range field, the program updates the view ending time to reflect the newly selected view range.
- *Selection area coordinate fields* — Indicate the exact time or time range currently selected in the canvas. Reading from left to right, these fields display the starting time, the ending time and the total time span selected. Selection coordinates may also be entered into any of the selection fields. After entering new coordinates, press **Enter** to update the selection area.
- *Total time* — Displays the time period of the entire data file in the time unit listed in the ticks display field.
- *Status bar* — Displays system information such as error messages and application information.

- *Toolbar* — Provides shortcuts to the most commonly used functions.
  - — Opens the **Read From** dialog to read in a data file. Browse to the desired data file and click **Read**. Note that the EventAnalyzer never writes to the data file, so when a data file is closed, the program does not prompt you to save changes. However, the program will prompt you to save changes affecting the configuration file (for example, changes to the viewing options made in the Legend).
  - and — Modify the range displayed in the canvas to show more or less time. When zooming in, one half the existing range will be displayed; when zooming out, twice the range will be displayed. The center of the screen remains constant when zooming. You can zoom out to display the entire data file or zoom in to a range as small as 0.001 nanoseconds.
  - — Adjusts the canvas to display only the selected range. To select a range, click the left mouse button and hold it down while moving your cursor horizontally across the canvas. Then click this button and the canvas will display the same range as the selection area.
  - — Changes the unit of measure in the ticks display field, the coordinates field, the selection area coordinates field, and the time scale. With each click of this button, the EventAnalyzer chooses a different measurement unit (Second, Millisecond, Microsecond, or Nanosecond). By default, the EventAnalyzer displays an appropriate time unit whenever the view is changed. Clicking the change time unit icon overrides selections made by the Auto Adjust Time Unit feature (see “Time Unit Settings” on page 104).
  - — Displays the **Legend** window describing the various labels used for each status indicator and recorded thread. Line colors, line styles, and icons can be modified (see “Using the Legend” on page 96).
  - — Provides advanced controls for searching events, states, and context switches (see “Search for Event, Status, and Context Switches” on page 101).
  - — This searching feature is not available for ThreadX.
  - *Browse history buttons* — Navigate to the earliest ( ), previous ( ), next ( ), and latest ( ) views in your view history. The EventAnalyzer stores up to 50 views. These buttons function like Back and Forward buttons in a web browser.
  - — This feature is not available with ThreadX.

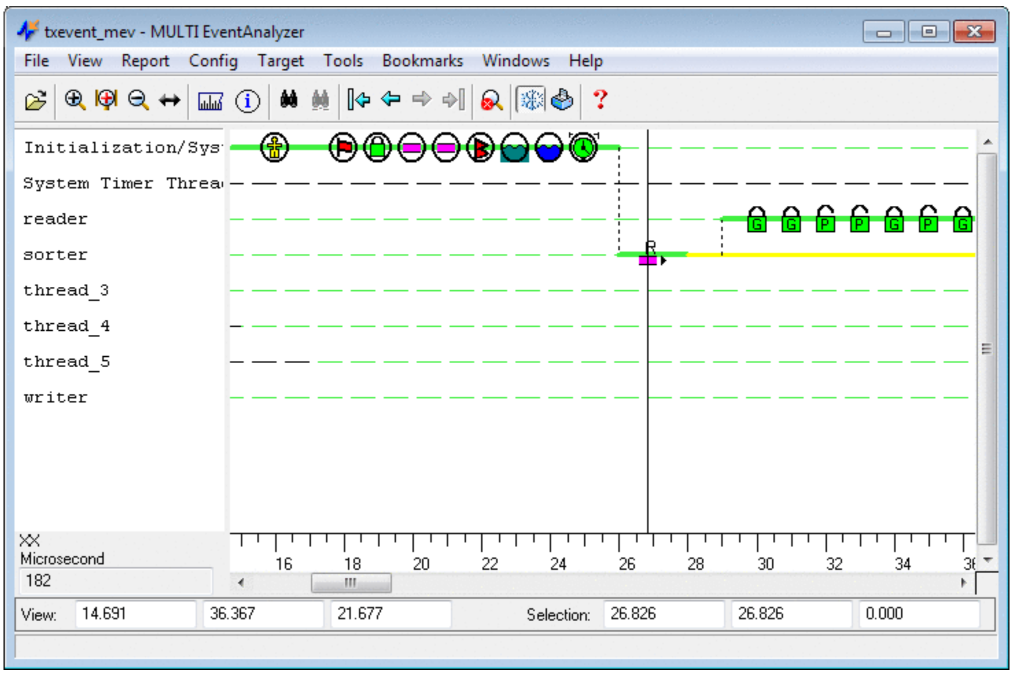
-  — Launches online help.

Selecting Data

This section describes how to read the EventAnalyzer display, how to display additional data about threads, events, and states, and how to customize the EventAnalyzer display.

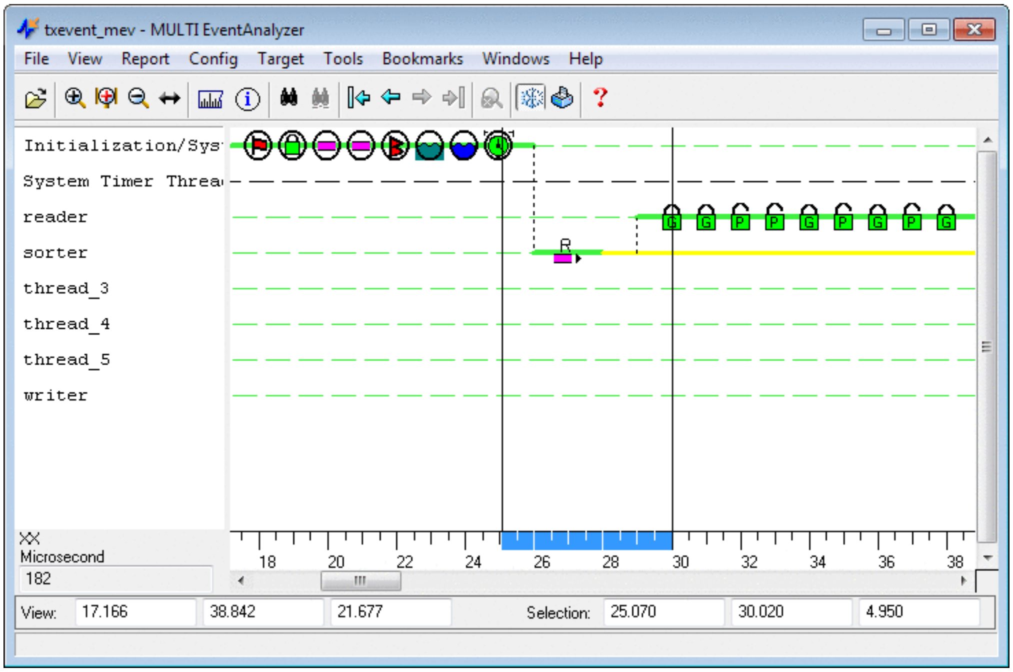
Selecting a Point in Time

To select a point in time, click the mouse pointer within the canvas. A solid vertical line appears indicating a Point in Time selection. When a single point in time is selected, the selection area coordinates beginning and ending fields contain the same value, and the range field is zero.



Selecting a Range of Time

To select a range of time, click the left mouse button and hold it down while moving your cursor horizontally across the canvas. When a range of time is selected, the selection area coordinates beginning and ending values contain different values and the range field indicates the amount of time selected.



Zooming to a Range Selection

To display only the selected range in the canvas, choose **View → Zoom To Range** or click after a range of time has been selected. The canvas adjusts to display only the selected range.

Creating a Reference Line

To create a temporary reference line in the canvas, press the **Shift** key and click the mouse pointer. This displays a vertical line that can be useful in analyzing event data. The line remains for as long as the left mouse button is pressed. The temporary reference line will not cancel a point in time or range of time selection.

Jumping to a Time Selection

The following mouse commands bring the mouse pointer to a selection. This is useful for quickly returning to a selection when, after scrolling, that time selection no longer appears within the view range.

- **Shift**+**Right-click** — Jumps to a point in time selection or to the start of a range of time selection. If the shortcut menu appears, it can be cleared with a **Shift**+**Left-click**.
- **Ctrl**+**Right-click** — Jumps to a point in time selection or to the end of a range of time selection. If the shortcut menu appears, it can be cleared with **Shift**+**Left-click**.

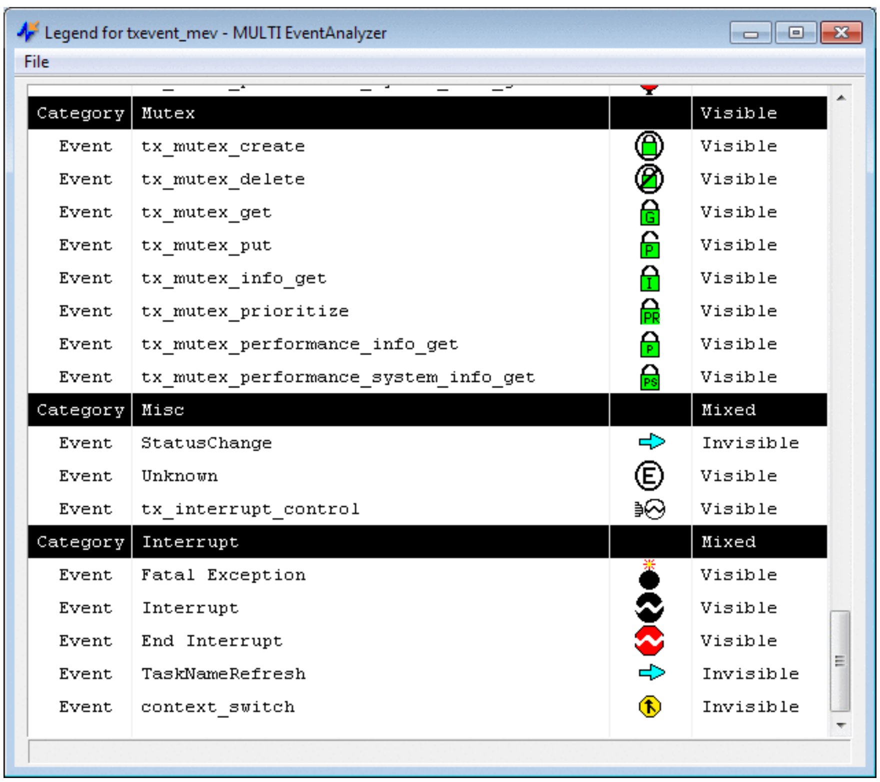
Viewing Event Data

This section describes how to customize the display of event data to show the events of interest, hide events that are not of interest, and view details about specific events.

Using the Legend

The Legend provides a reference for displaying and modifying the meaning of the various line colors, line styles and icons appearing in the canvas.

To open the legend, select **Config** → **Legend** or click . The ThreadX Events Legend appears:



The legend displays the names of all system events and thread states, the icon or line style displayed for each event or status, and the visibility attribute for each event or status. The event/status list is sorted by categories. In the Legend depicted above, `Mutex`, `Misc`, and `Interrupt` are categories.

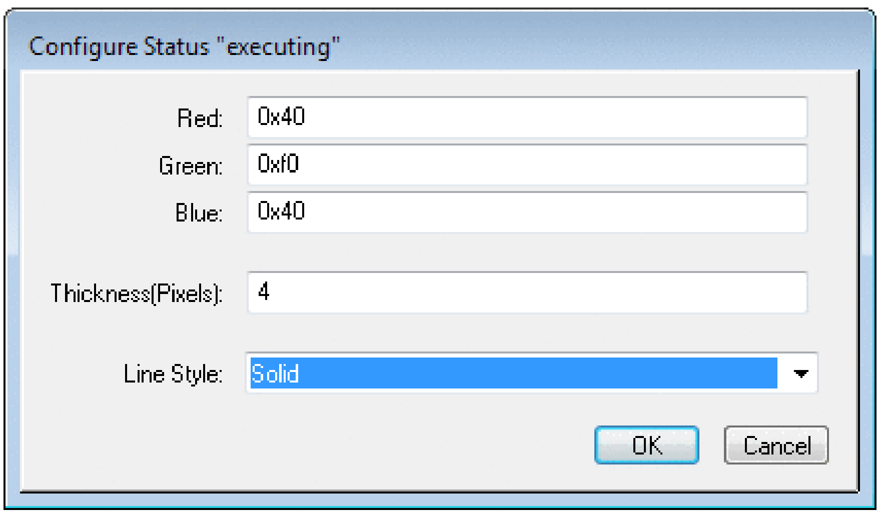
Note

Categories are defined in the configuration file. For information about defining categories, see Chapter 13, “EventAnalyzer Configuration Files” on page 107.

The Legend lists each status first and displays an example segment of its associated line color and style.

To change an event icon, click the icon to view the MULTI internal icon library. Select the desired replacement and click **OK**.

To modify line color and style, click the example line segment in the **Legend** window. The line configuration (RGB) window appears:



The Red, Green, and Blue fields determine the line color by the conventional RGB color scheme, with values ranging from 0 to 255 for the intensity of each color. Choosing the values Red 0, Green 0, Blue 0 would result in black. The RGB values can be specified in hexadecimal (prefixed by `0x`) or decimal.

Select the desired line thickness and choose a line style from the drop-down list. Click **OK** and the Legend appears with the new line color and/or style.

The last column of the **Legend** window displays the visibility attribute of each event or status. Click the word to toggle the attribute between Visible and Invisible. Invisible items do not appear in the canvas. To toggle the visibility attribute for an entire category of events, click the visibility attribute of the category. Changes made to event or status visibility do not actually change the data file; the invisibility attribute only determines whether that event or status will be displayed in the EventAnalyzer canvas.

The EventAnalyzer can save changes made in the EventAnalyzer legend to the configuration file. If changes were made in the Legend, the EventAnalyzer will ask whether to save or discard changes upon exiting the program.

View Event, and Status and Thread Details

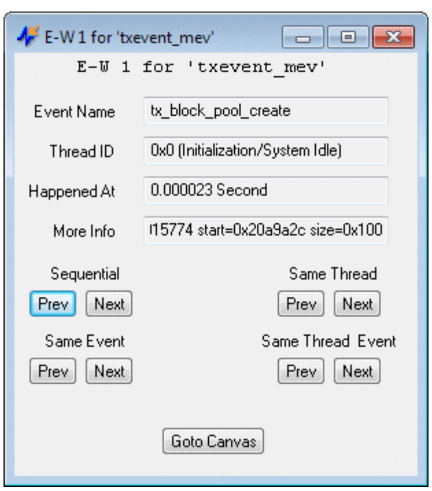
You can double-click any icon or line in the canvas to open the object view displaying detailed information about the event or status. You can also right-click an event or status and choose **View Object**, **Zoom In**, **Zoom Out**, or (if a range has been selected) **Zoom into Selection**. You can double-click any thread in the thread name region to open the **Thread Info** dialog box displaying detailed information about the thread.

The object view provides detailed information about a particular event, status, or thread as well as advanced search capabilities to allow more ways to analyze and troubleshoot the target system.

Multiple object views can be opened simultaneously. Choose **View → Delete Object Views** to close all object views as well as any open **Search Results** windows.

Viewing Event Details

The **Event View** dialog box shows details of a specific event:



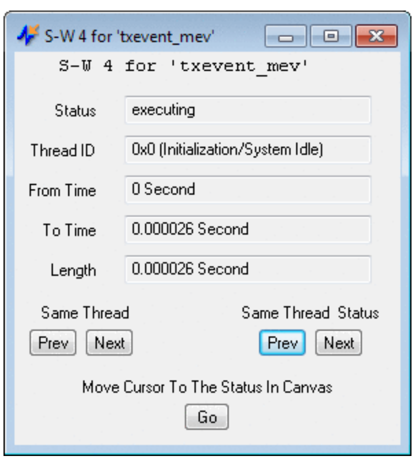
This dialog box shows the **Event Name**, **Thread ID**, and the time at which the event occurred. The **More Info** field displays extra data about the event. For a list of the extra data associated with all standard ThreadX services, see Chapter 14, “ThreadX Services Reference” on page 119. For information about defining and logging custom events and displaying extra event information for them, see “Defining Events” on page 110.

Use the following buttons in the dialog to browse to the object view for adjacent events:

- **Sequential** — Shows the next/previous event of any type for any thread.
- **Same Event** — Shows the next/previous event of the same type for any thread.
- **Same Thread** — Shows the next/previous event of any type for the same thread.
- **Same Thread/Event** — Shows the next/previous event of the same type for the same thread.
- **Goto Canvas** — Brings the EventAnalyzer canvas to the foreground and places the mouse pointer on the event icon.

Viewing Status Details

The **Status View** dialog box shows details of a status line:



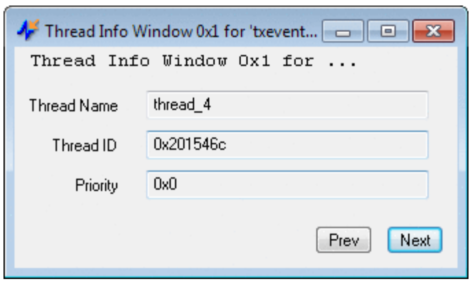
This dialog box shows the **Status Name**, the **Thread ID** of the thread to which the status applies, the times at which the status started and ended, and the duration (**Length**) of that status.

Use the following buttons to navigate to the Status View for other states.

- **Same Thread** — Shows the next/previous status of any type for the same system thread.
- **Same Thread/Status** — Shows the next/previous status of the same type for the same thread.
- **Go** — Brings the EventAnalyzer canvas to the foreground and places the mouse pointer on the status line.

Viewing Thread Details

The **Thread Info** dialog box shows details of a particular thread:



This dialog box shows the **Thread Name**, the **Thread ID**, and the priority of the thread.

Use the **Prev** and **Next** buttons to scroll through all threads in the order they appear in the thread name region.

Viewing Context Switch Details

To view details of a context switch, double-click the context switch object in the canvas. The object view for a context switch displays the thread from which the system changed, the thread to which it changed, and the time at which the change occurred.

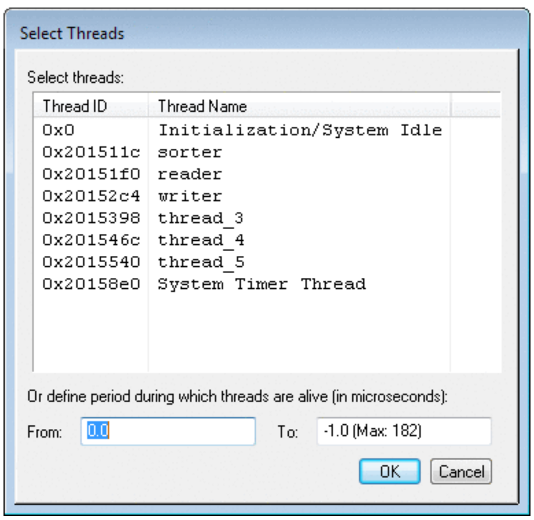
Use the **Prev** and **Next** buttons to jump sequentially between context switches throughout the data file.

Click **Go** to bring the EventAnalyzer canvas to the foreground and place the mouse pointer on the context switch.

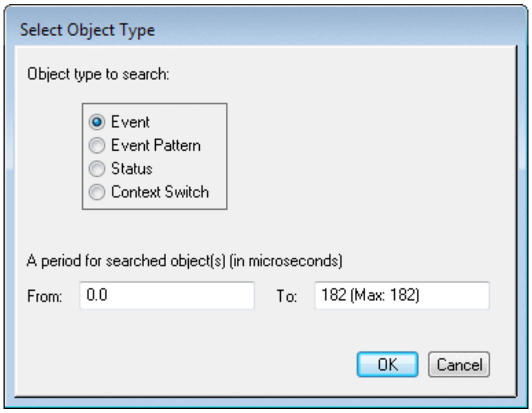
Search for Event, Status, and Context Switches

The advanced search features of the EventAnalyzer enable you to scan the event data for all instances of any particular event, status, or context switch. You can restrict the search to a specific time range. The EventAnalyzer displays a list of matching events that can be used to control the focus of the canvas. Selecting an item from the list causes the canvas to jump to that item.

To use the search feature, select **View** → **Search** (or click ) to open the **Select Threads** dialog box:



This dialog box lists all the threads recorded in the data file. Select the thread or threads on which to search. Click **OK** to open the **Select Object Type** dialog box:


Select Object Type
Object type to search:
Event
Event Pattern
Status
Context Switch
A period for searched object(s) (in microseconds)
From:
0.0
To:
182 (Max: 182)
OK
Cancel

Choose one of the following options, specify a time range, if desired, and click **OK** to start the search:

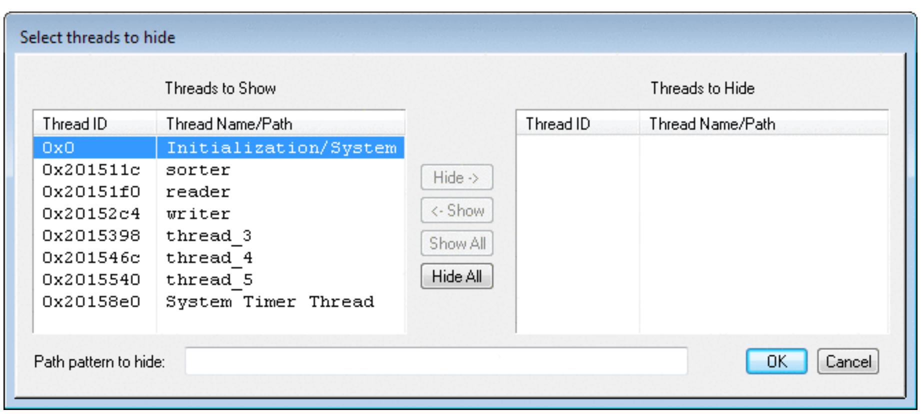
- **Event**, **Event Pattern**, or **Status** — Opens another dialog in which you must specify the event, event pattern, or status for which to search. When you click **OK** in this dialog, the **Search Results** window will open and display the results of the search. Click any item in the list and the canvas jumps to that event, event pattern, or status.
- **Context Switch** — Opens the **Search Results** window and displays a list of all context switches for the selected thread. Click any item in the list and the canvas jumps to that context switch.

Multiple **Search Results** windows can be open simultaneously. Choose **View** → **Delete Object Views** to close all **Search Results** windows as well as any open object view windows.

Changing the Hidden Task List

The MULTI EventAnalyzer allows you to *hide* selected threads so that they do not appear in the canvas, even though events and status changes for the thread have been logged.

To hide threads, choose **View** → **Change Hidden Thread List** to open the Hidden Task list:



The **Displayed Threads** are displayed on the left side of the window and the **Hidden Threads** are displayed on the right. Select threads from either list, then click **Hide** or **Show**.

The **Show All** button moves all threads to the **Displayed Threads** list. The **Hide all** button moves all threads to the **Hidden Threads** list.

Click **OK** to apply your changes.

Generating Reports

The EventAnalyzer generates a variety of reports in user-defined time frames.

A report displays the number and percentage of occurrences of each event compared to the total number of events in the time frame. For thread status reports, the time spent in each status is displayed in time units and as a percentage of the total time frame. This can be used to show how much time threads spend executing and in the other states.

The reports are available from the **Report** menu.

Configuration Menu Operations

This section describes how to change the **Canvas name** and the time units used in the canvas.

For information about changing the format of lines and icons in the canvas, see “Using the Legend” on page 96.

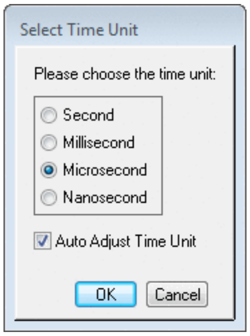
Changing the Canvas Name

To change the canvas name (displayed in the title bar of the EventAnalyzer), select **Config → Canvas name**, enter the new canvas name, and click **OK**. The new canvas name appears in the main screen.

Time Unit Settings

The unit of measure used to display times in the canvas can be seconds, milliseconds, microseconds, or nanoseconds. The EventAnalyzer's Auto Adjust Time Unit feature selects a time unit appropriate to the amount of data displayed in the canvas.

To enable or disable the Auto Adjust Time Unit feature, select **Config → Time Unit**, select or clear the **Auto Adjust** check box, and click **OK**.



You can override the **Auto Adjust Time Unit** selection at any time by clicking the **Time Unit** icon on the toolbar one or more times to iterate through the available unit options.

Chapter 13

EventAnalyzer Configuration Files

Contents

An EventAnalyzer configuration file lists every event or status native to the ThreadX RTOS and defines display properties such as the status line colors or event icons.

Some display parameters, such as the thread status line colors or event icons, can be changed either by using the EventAnalyzer **Legend** window (see "Using the Legend" on page 96) or by modifying the configuration file. Other parameters (such as the icon used to indicate overlapping events) can only be changed by editing the configuration file.

If your application logs user-defined events, the configuration file must be updated so that the EventAnalyzer recognizes those events and includes the necessary event data. For information about adding user-defined events, see "User-Defined Events" on page 82.

The initial installation of MULTI contains a complete EventAnalyzer configuration file set to default conditions. No modifications to the configuration file are required to use the EventAnalyzer. To customize the display, the configuration file can be edited. This chapter describes each parameter contained in the configuration file. Except where noted, these parameters can also be defined using the EventAnalyzer.

The configuration file defines the following ThreadX events:

- Thread Status
- Event
- Event Category
- Overlap

You can modify the configuration file to include other events, as necessary, or to change display parameters of these previously defined events.

Each line of the configuration file describes a single event. The following sections describe the formatting conventions of each object type. The configuration file format is case-insensitive.

The configuration file **threadx.mc** contains the ThreadX-specific configuration. When the EventAnalyzer is invoked, it attempts to locate a user-specific version of this file in:

- Windows — ***user_dir*\Application Data\GHS\event_analyzer**

- Linux/Solaris — ***user_dir*/.ghs/event_analyzer/**

If a user-specific version of **threadx.mc** is not available, **mevgui** searches for the file in the **defaults\event_analyzer** subdirectory of the MULTI installation. Any manual edits to the file located in the MULTI installation affect all users of the installation who do not have a user-specific version of **threadx.mc**. Any changes saved via the GUI cause the user-specific file to be created or updated, and do not affect other users.

Thread Status

The format for defining a thread status is as follows:

`MEV_Object:MEV_Status:`*`id`*`:`*`status_name`*`:`*`rgb`*`:`*`style`*`:`*`thickness`*`:`*`visibility`*

where:

- `MEV_Object` and `MEV_Status` — Are keywords and must be entered as shown above for all thread status settings.
- *`id`* — Is an integer used to identify the thread status on the target.
- *`status_name`* — Is a string, such as `ready`, `executing`, or `terminated`, corresponding to the thread status.
- *`rgb`* — Determines the status line color. Enter the hexadecimal value representing the desired color.
- *`style`* — Refers to the line style used to represent the status of the thread in the EventAnalyzer **Canvas**. The available line styles are:
  - `MEV_Solid`
  - `MEV_Dot`
  - `MEV_Dash`
  - `MEV_DashDot`
  - `MEV_DashDotDot`
- *`thickness`* — Determines the thickness of the line in pixels.
- *`visibility`* — Indicates whether the status will be displayed in the EventAnalyzer **Canvas**. Enter either `MEV_Visible` (the default) or `MEV_Invisible`.

Defining Events

The format for defining an event is:

```
MEV_Event:type:sub_type:event_name:icon_name:[extra_data:]visibility
```

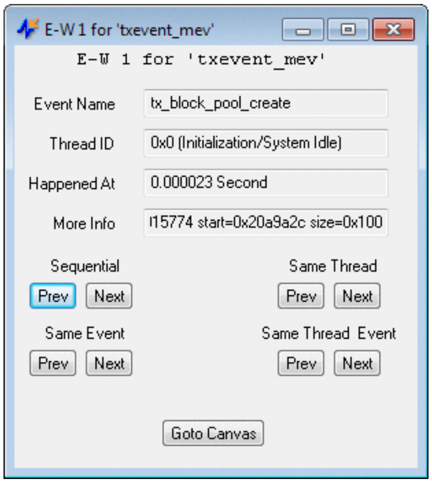
where:

- `MEV_Event` — Is a keyword and should be entered as shown above.
- *type* and *sub_type* — Are two numbers used by ThreadX to identify an event. For example a *type* of `0x3` and a *sub_type* of `0x16` corresponds to the `tx_thread_create` event.
- *event_name* — Is the name of the event, such as `tx_byte_allocate`, `tx_event_flags_create`, or `tx_semaphore_delete`.
- *icon_name* — Is the name of the icon or the filename of an external graphic icon file used to represent the event in the EventAnalyzer. MULTI provides many built-in icons. The EventAnalyzer **Legend** window displays a list of these icons and the icon names. To specify one of these icons, enter the icon's name. Do not include a **.bmp** file extension.

  You can also use an icon other than one provided by MULTI, by specifying an external graphic icon filename. The file must be in the **.bmp** format. Enter the filename of the graphic icon file, including the **.bmp** file extension. Enclose the filename in quotation marks if it contains whitespace or a colon.
- *extra_data* — Specifies optional, additional data that can be useful in understanding an event. For example, if a semaphore is created, the new `Semaphore ID` can be recorded as extra data for the logged semaphore create event. Subsequent operations on the semaphore can also log the `Semaphore ID` so that operations on the same semaphore can be located easily. (See the next section for more information.)
- *visibility* — Indicates whether the status will be displayed in the EventAnalyzer **Canvas**. Enter either `MEV_Visible` (default) or `MEV_Invisible`.

Specifying Extra Data

The event window displays extra data relating to the selected event as defined in the configuration file. When an event occurs, the system logs the standard event data (event name, thread ID, and elapsed time) and any extra data defined. This extra data should be described by the configuration file. All of this data can then be viewed in an EventAnalyzer event window, as pictured below.



The format for the *`extra_data`* field entry is the character sequence `MEV_Extra=` followed by a string enclosed in quotation marks.

Format strings of the following types are permitted:

- `%C` — Indicates that the next data item is a 1-byte character value.
- `%1D` — Indicates that the next data item is a 1-byte integer displayed as a decimal value.
- `%1X` — Indicates that the next data item is a 1-byte integer displayed as a hexadecimal value.
- `%2D` — Indicates that the next data item is a 2-byte integer displayed as a decimal value.
- `%2X` — Indicates that the next data item is a 2-byte integer displayed as a hexadecimal value.
- `%4D` — Indicates that the next data item is a 4-byte integer displayed as a decimal value.
- `%4X` — Indicates that the next data item is a 4-byte integer displayed as a hexadecimal value.
- `%S` — Indicates that the next data item is an array of a basic type. Arrays can be specified as:

```
array_length%Sarray_element
```

where:

- *array_length* is a data item that indicates the length of the array.
- *array_element* is a data item that indicates the type of each array element.

For example, a character string whose length is specified as a 4-byte integer is expressed as:

```
%4X%S1C
```

Some examples of extra data configurations follow:

```
...:MEV_Extra="queue_ptr=%4X":...
...:MEV_Extra="semaphore_ptr=%4X initial_value=%4X":...
```

Event Categories

The EventAnalyzer allows related events to be grouped into categories. The format for defining an event category is:

MEV_Event_Category:*category_name*:*visibility*

where:

- MEV_Event_Category — Is a keyword and should be entered in the configuration file as shown above.
- *category_name* — Is the name of the category as defined by the user. Some examples of event categories in the ThreadX kernel are Block Pool, Event Flags, and Queue.
- *visibility* — Indicates whether or not the threads within the category will be displayed in the EventAnalyzer **Canvas**. If not defined, the default value is MEV_Visible. Categories that include visible and invisible objects require no visibility distinction. The attribute selected at the category level applies to all the events in that category. Category and object visibility can also be modified from the **Legend** window.

Below the category definition line, list each of the events to be included in that category. For example:

```
MEV_Event_Category:Thread:MEV_Visible
MEV_Event:0x3:0x16:tx_thread_create:tx_t_create:MEV_Extra="thread_ptr=
  %4X statck_start=%4X stack_size=%4X priority=%4X":MEV_Visible
MEV_Event:0x3:0x17:tx_thread_delete:tx_t_delete:MEV_Extra="thread_ptr=
  %4X":MEV_Visible
```

All events following an event category definition are considered to be part of the event category, until another event category is defined.

An event can be included in multiple event categories. To do this, include a description line for the event in each category. In case of conflicting event definitions, the event description line appearing last in the configuration file determines the visibility attribute of the object.

Unknown Events

The EventAnalyzer employs an `unknown` object to represent any events or states not defined, or not defined fully in the ThreadX configuration file.

You can also specify `unknown` as the type for any object, for example:

```
MEV_Object:MEV_Status:0xa:unknown:0xff0000:MEV_Solid:5:MEV_Visible
MEV_Event:0x0:0x0:unknown:questionmark:MEV_Visible
```

Miscellaneous Configuration Options

Event Overlap Icon

If two events occur within a short amount of time, the event icons may overlap in the display and, as a result, be difficult to read. By modifying the overlap setting, the EventAnalyzer can display a single icon indicating that two or more icons overlap. The correct event icons will be shown when the display is expanded to a resolution at which the icons no longer overlap

Using the overlap feature increases the redraw speed in the canvas.

By default, event overlap is not enabled. It can be enabled by setting the icon as described below. The format is as follows:

```
MEV_Misc:MEV_Overlap:icon_name
```

The **Legend** window displays all the available icons and their filenames. Select an appropriate icon from the **Legend** and enter its name in the `icon_name` section of the overlap icon definition in the configuration file.

The overlap icon definition is a special entry in the configuration file that cannot be changed using the **Legend** window at run time. Therefore, this feature must be enabled or disabled prior to starting the EventAnalyzer.

Status Line Position

If the `MEV_Center_Status` option is true, status lines are vertically centered behind event icons. If this option is false, status lines are displayed below event icons. This option defaults to false, but the default configuration file sets it to true.

The format for this option is:

```
MEV_Misc:MEV_Center_Status:true|false
```

Tick Value Display

If the `MEV_Tick_Pattern` option is true, common digits in tick values are displayed in the ticks display field, as described in "The EventAnalyzer Window" on page 89. If this option is false, the full tick value is displayed in the canvas. This option defaults to true.

The format for this option is:

```
MEV_Misc:MEV_Tick_Pattern:true|false
```

Warning for Unused Extra Data

If the `MEV_Unused_Extra_Data_Warning` option is true, a warning is printed when an event contains *`extra_data`* that is not specified in the corresponding `MEV_Event` entry. If this option is false, no warning is printed. This option defaults to true.

The format for this option is:

```
MEV_Misc:MEV_Unused_Extra_Data_Warning:true|false
```

Warning for Missing Extra Data

If the `MEV_Extra_Data_Warning` option is true, a warning is printed when an `MEV_Event` entry specifies *`extra_data`* that is not contained in the event. If this option is false, no warning is printed. This option defaults to true, but the default configuration file sets it to false.

The format for this option is:

```
MEV_Misc:MEV_Extra_Data_Warning:true|false
```

Reserved Keywords

The EventAnalyzer configuration file reserves the following keywords:

- `MEV_Object`
- `MEV_Event_Category`
- `MEV_Event`
- `MEV_Status`
- `MEV_Misc`
- `Context Switch`
- `running`
- `MEV_Visible`
- `MEV_Invisible`
- `unknown`

- `MEV_Refresh_Interval`

Chapter 14

ThreadX Services Reference

Contents

This chapter lists each ThreadX service in alphabetical order and provides the extra data included for that service. For example, when a semaphore operation is performed, the extra data indicates which semaphore is being operated upon.

Memory Block Pool Services

- — `tx_block_allocate`:
    - **Pool Address** — Address of the block pool control block
    - **Pointer Address** — Address of the return block pointer
    - **Block Address** — Address of the block allocated
- — `tx_block_pool_create`:
    - **Pool Address** — Address of the block pool control block
    - **Pool Memory Area Address** — Address of the block pool memory area
    - **Pool Size** — Number of bytes in the block pool
- — `tx_block_pool_delete`:
    - **Pool Address** — Address of the block pool control block
- — `tx_block_pool_info_get`:
    - **Pool Address** — Address of the block pool control block
- — `tx_block_pool_performance_info_get`:
    - **Pool Address** — Address of the block pool control block
- — `tx_block_pool_performance_system_info_get`:
    - (No extra information)
- — `tx_block_pool_prioritize`:
    - **Pool Address** — Address of the block pool control block
- — `tx_block_release`:
    - **Pool Address** — Address of the block pool control block
    - **Block Address** — Address of the block being released

Memory Byte Pool Services

- — `tx_byte_allocate`:
  - **Pool Address** — Address of the byte pool control block
  - **Pointer Address** — Address of the return memory pointer
  - **Request Size** — Number of bytes in the request
  - **Memory Address** — Address of the memory being allocated
- — `tx_byte_pool_create`:
  - **Pool Address** — Address of the byte pool control block
  - **Pool Memory Area Address** — Address of the byte pool memory area
  - **Pool Size** — Number of bytes in the byte pool memory area
- — `tx_byte_pool_delete`:
  - **Pool Address** — Address of the byte pool control block
- — `tx_byte_pool_info_get`:
  - **Pool Address** — Address of the byte pool control block
- — `tx_byte_pool_performance_info_get`:
  - **Pool Address** — Address of the byte pool control block
- — `tx_byte_pool_performance_system_info_get`:
  - (No extra information)
- — `tx_byte_pool_prioritize`:
  - **Pool Address** — Address of the byte pool control block
- — `tx_byte_release`:
  - **Pool Address** — Address of the byte pool control block
  - **Memory Address** — Address of the memory being released

Event Flags Services

- — `tx_event_flags_create`:
    - **Event Group** — Pointer to event flags group control block
- — `tx_event_flags_delete`:
    - **Event Group** — Pointer to event flags group control block
- — `tx_event_flags_get`:
    - **Event Group** — Pointer to event flags group control block
    - **Requested Flags** — Flags requested for the get operation
    - **Option** — Get option that was specified
- — `tx_event_flags_info_get`:
    - **Event Group** — Pointer to event flags group control block
- — `tx_event_flags_performance_info_get`:
    - **Event Group** — Pointer to event flags group control block
- — `tx_event_flags_performance_system_info_get`:
    - (No extra information)
- — `tx_event_flags_set`:
    - **Event Group** — Pointer to event flags group control block
    - **Flags** — Flags to apply to the event flags group
    - **Option** — Set option that was specified
- — `tx_event_flags_set_notify`:
    - **Event Group** — Pointer to event flags group control block
    - **Notify** — Pointer to notification callback function

Interrupt Services

- — `tx_interrupt_control`:
  - **New Posture** — New interrupt posture to apply

Mutex Services

- — `tx_mutex_create`:
  - **Mutex Pointer** — Pointer to the mutex control block
  - **Priority Inheritance** — Priority inheritance setting
- — `tx_mutex_delete`:
  - **Mutex Pointer** — Pointer to the mutex control block
- — `tx_mutex_get`:
  - **Mutex Pointer** — Pointer to the mutex control block
  - **Owner** — Pointer to the thread control block of the mutex owner
  - **Ownership Count** — Current mutex ownership count
- — `tx_mutex_info_get`:
  - **Mutex Pointer** — Pointer to the mutex control block
- — `tx_mutex_performance_info_get`:
  - **Mutex Pointer** — Pointer to the mutex control block
- — `tx_mutex_performance_system_info_get`:
  - (No extra information)
- — `tx_mutex_prioritize`:
  - **Mutex Pointer** — Pointer to the mutex control block
- — `tx_mutex_put`:
  - **Mutex Pointer** — Pointer to the mutex control block
  - **Owner** — Pointer to the thread control block of the mutex owner
  - **Ownership Count** — Current mutex ownership count

Message Queue Services

- — `tx_queue_create`:
  - **Queue Pointer** — Pointer to the queue control block
  - **Queue Address** — Address of the start of the queue memory area
  - **Queue Size** — Size of queue memory area
  - **Message Size** — Size of queue messages
- — `tx_queue_delete`:
  - **Queue Pointer** — Pointer to the queue control block
- — `tx_queue_flush`:
  - **Queue Pointer** — Pointer to the queue control block
- — `tx_queue_front_send`:
  - **Queue Pointer** — Pointer to the queue control block
  - **Source Address** — Pointer to the message to send
- — `tx_queue_info_get`:
  - **Queue Pointer** — Pointer to the queue control block
- — `tx_queue_performance_info_get`:
  - **Queue Pointer** — Pointer to the queue control block
- — `tx_queue_performance_system_info_get`:
  - (No extra information)
- — `tx_queue_prioritize`:
  - **Queue Pointer** — Pointer to the queue control block
- — `tx_queue_receive`:
  - **Queue Pointer** — Pointer to the queue control block
  - **Destination Address** — Pointer to the receive message destination
- — `tx_queue_send`:
  - **Queue Pointer** — Pointer to the queue control block
  - **Source Address** — Pointer to the message to send
- — `tx_queue_send_notify`:
  - **Queue Pointer** — Pointer to the queue control block

- **Notify** — Pointer to notification callback function

Semaphore Services

- — `tx_semaphore_ceiling_put`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
  - **Current Count** — Current semaphore count
  - **Ceiling** — Maximum limit
- — `tx_semaphore_create`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
  - **Initial Count** — The initial semaphore count
- — `tx_semaphore_delete`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
- — `tx_semaphore_get`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
  - **Current Count** — Current semaphore count
- — `tx_semaphore_info_get`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
- — `tx_semaphore_performance_info_get`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
- — `tx_semaphore_performance_system_info_get`:
  - (No extra information)
- — `tx_semaphore_prioritize`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
- — `tx_semaphore_put`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
  - **Current Count** — Current semaphore count
- — `tx_semaphore_put_notify`:
  - **Semaphore Pointer** — Pointer to the semaphore control block
  - **Notify** — Pointer to notification callback function

Thread Services

- — `tx_thread_create`:
  - **Thread Pointer** — Pointer to the thread control block
  - **Stack Starting Address** — Starting memory address of the thread's stack
  - **Stack Size** — Size of thread’s stack in bytes
  - **Priority** — Thread’s priority
- — `tx_thread_delete`:
  - **Thread Pointer** — Pointer to the thread control block
- — `tx_thread_entry_exit_notify`:
  - **Thread Pointer** — Pointer to the thread control block
  - **Notify** — Pointer to notification callback function
- — `tx_thread_identify`:
  - (No extra information)
- — `tx_thread_info_get`:
  - **Thread Pointer** — Pointer to the thread control block
- — `tx_thread_performance_info_get`:
  - **Thread Pointer** — Pointer to the thread control block
- — `tx_thread_performance_system_info_get`:
  - (No extra information)
- — `tx_thread_preemption_change`:
  - **Thread Pointer** — Pointer to the thread control block
  - **Previous Threshold** — Old preemption threshold
  - **New Threshold** — New preemption threshold
- — `tx_thread_priority_change`:
  - **Thread Pointer** — Pointer to the thread control block
  - **Previous Priority** — Old priority
  - **New Priority** — New priority
- — `tx_thread_relinquish`:
  - (No extra information)

- — `tx_thread_reset`:
  - **Thread Pointer** — Pointer to the thread control block to reset
- — `tx_thread_resume`:
  - **Thread Pointer** — Pointer to the thread control block to resume
- — `tx_thread_sleep`:
  - **Ticks** — Number of ticks to sleep
- — `tx_thread_stack_error_notify`:
  - **Notify** — Pointer to notification callback function
- — `tx_thread_suspend`:
  - **Thread Pointer** — Pointer to the thread control block to suspend
- — `tx_thread_terminate`:
  - **Thread Pointer** — Pointer to the thread control block to terminate
- — `tx_thread_time_slice_change`:
  - **Thread Pointer** — Pointer to the thread control block
  - **Previous Time-slice** — Thread's old time-slice
  - **New Time-slice** — Thread's new time-slice
- — `tx_thread_wait_abort`:
  - **Thread Pointer** — Pointer to the thread control block

Application Timer Services

- — `tx_time_get`:
  - **Current Time** — Current time (tick) count
- — `tx_time_set`:
  - **New time** — New time (tick) count
- — `tx_timer_activate`:
  - **Timer Pointer** — Pointer to the timer control block
- — `tx_timer_change`:
  - **Timer Pointer** — Pointer to the timer control block
  - **Initial Ticks** — Number of ticks before initial expiration
  - **Reschedule Ticks** — Number of ticks for subsequent expirations
- — `tx_timer_create`:
  - **Timer Pointer** — Pointer to the timer control block
  - **Initial Ticks** — Number of ticks before initial expiration
  - **Reschedule Ticks** — Number of ticks for subsequent expirations
  - **Activate** — Auto-activation selection
- — `tx_timer_deactivate`:
  - **Timer Pointer** — Pointer to the timer control block
- — `tx_timer_delete`:
  - **Timer Pointer** — Pointer to the timer control block
- — `tx_timer_info_get`:
  - **Timer Pointer** — Pointer to the timer control block
- — `tx_timer_performance_info_get`:
  - **Timer Pointer** — Pointer to the timer control block
- — `tx_timer_performance_system_info_get`:
  - (No extra information)

Index

P

Q

S

T