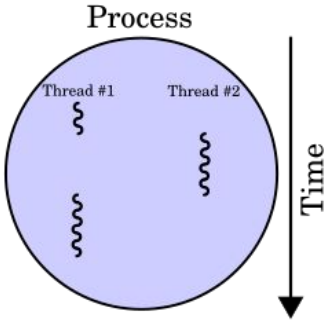
THREADX

sea|me

software engineering in automotive
and mobility ecosystems

THREAD



Thread vs Process

- A **thread** is the smallest unit of execution inside a process.
- A **process** is an independent program with its own memory space.
- Threads **share** the same memory and resources; processes **do not**.
- Thread communication is **faster** but less isolated.
- Threads are **lightweight**, while processes are **heavier** to create and manage.

sea|me
software engineering in automotive
and mobility ecosystems

STM32U585

ThreadX can *manage* many threads but **STM32U585** (which uses an **Arm Cortex-M33** core) has **only one CPU core**.

That means:

- Only **one thread** t.
- The other threads are in states like *ready*, *block* d.
- The RTOS **rapidly switches** between them, giving the *illusion* of concurrency — this is called **time-slicing** or **preemptive multitasking**.

👉 You can create many threads — but **only one executes at a time**.

sea|me

software engineering in automotive and mobility ecosystems

THREADX EXECUTION

System Reset Vector: Initial hardware setup and reset process when the system powers on.

Entry Point: The starting point for program execution after the reset.

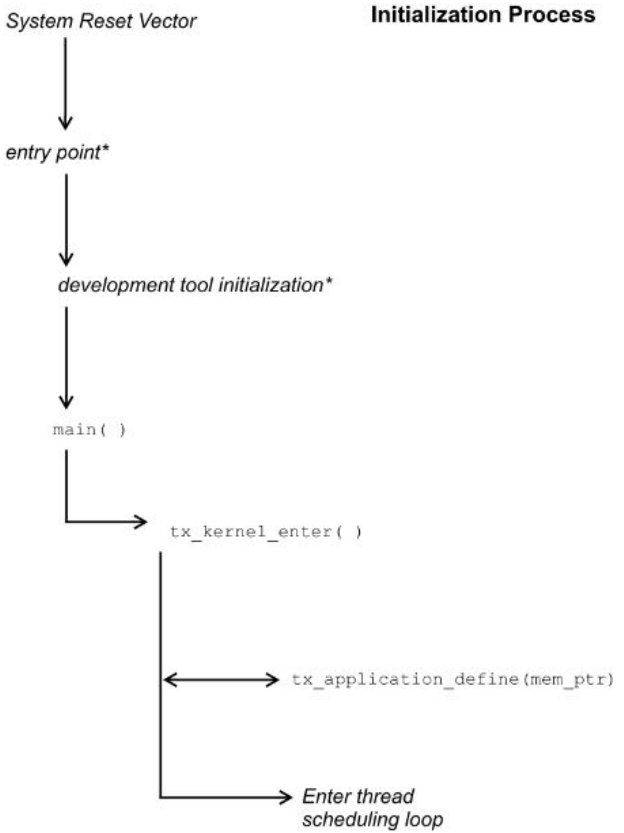
Development Tool Initialization: Initialize development tools for debugging and setup.

main(): The main function where the application's logic begins.

tx_kernel_enter(): Enter the ThreadX kernel to manage multitasking and system resources. (The **kernel** is the **core part of an operating system**, it acts as a **bridge** between software (applications) and hardware.)

tx_application_define(mem_ptr): Define the application and memory pointer for tasks to be scheduled by the kernel.

Enter Thread Scheduling Loop: The kernel starts managing and scheduling threads to run concurrently.



ThreadX Scheduling & Thread Starvation

- Schedules threads based on priority, executing the highest priority thread first. Threads of equal priority are executed in a FIFO (first-in-first-out) manner.
- The maximum number of ThreadX priorities can be configured from 32 to 1024 in increments of 32. The actual maximum is determined by the **TX_MAX_PRIORITIES** constant during compilation.
- Lower priority threads run only when no higher priority threads are ready.
- If a high priority thread is always ready, lower priority threads may never run.
- This issue is known as *thread starvation*.

```
UINT tx_thread_create(TX_THREAD *thread_ptr,
                      CHAR *name_ptr, VOID (*entry_function)(ULONG),
                      ULONG entry_input, VOID *stack_start,
                      ULONG stack_size, UINT priority,
                      UINT preempt_threshold, ULONG time_slice,
                      UINT auto_start)
```

```
TX_THREAD       my_thread;
UINT            status;


/* Create a thread of priority 15 whose entry point is
   "my_thread_entry". This thread's stack area is 1000
   bytes in size, starting at address 0x400000. The
   preemption-threshold is setup to allow preemption of threads
   with priorities ranging from 0 through 14. Time-slicing is
   disabled. This thread is automatically put into a ready
   condition.  */
status =  tx_thread_create(&my_thread, "my_thread_name",
                my_thread_entry, 0x1234,
                (VOID *) 0x400000, 1000,
                15, 15, TX_NO_TIME_SLICE,
                TX_AUTO_START);


/* If status equals TX_SUCCESS, my_thread is ready
   for execution!  */
```

sea|me

software engineering in automotive
and mobility ecosystems

```
/* Thread's entry function. When "my_thread" actually
   begins execution, control is transferred to this
   function. */
VOID my_thread_entry (ULONG initial_input)
{

    /* When we get here, the value of initial_input is
       0x1234. See how this was specified during
       creation.  */

    /* The real work of the thread, including calls to
       other function should be called from here!  */

    /* When this function returns, the corresponding
       thread is placed into a "completed" state.  */
}
```

sea|me

software engineering in automotive
and mobility ecosystems

thread_ptr	Pointer to a thread control block.
name_ptr	Pointer to the name of the thread.
entry_function	Specifies the initial C function for thread execution. When a thread returns from this entry function, it is placed in a *completed* state and suspended indefinitely.
entry_input	A 32-bit value that is passed to the thread's entry function when it first executes. The use for this input is determined exclusively by the application.
stack_start	Starting address of the stack's memory area.
stack_size	Number bytes in the stack memory area. The thread's stack area must be large enough to handle its worst-case function call nesting and local variable usage.
priority	Numerical priority of thread. Legal values range from 0 through (TX_MAX_PRIORITES-1), where a value of 0 represents the highest priority.

sea|me
software engineering in automotive and mobility ecosystems

🧵 Thread Preemption Behavior

A thread can only be preempted by threads with **priority ≤ preempt_threshold**.

If `preempt_threshold <= priority`: ✅ Normal preemption.

If `preempt_threshold < priority`: ❌ Only higher-priority threads **below the threshold** can preempt.

Example: priority = 10

preempt_threshold = 8

Threads with priority **0–8** ➜ Can preempt.

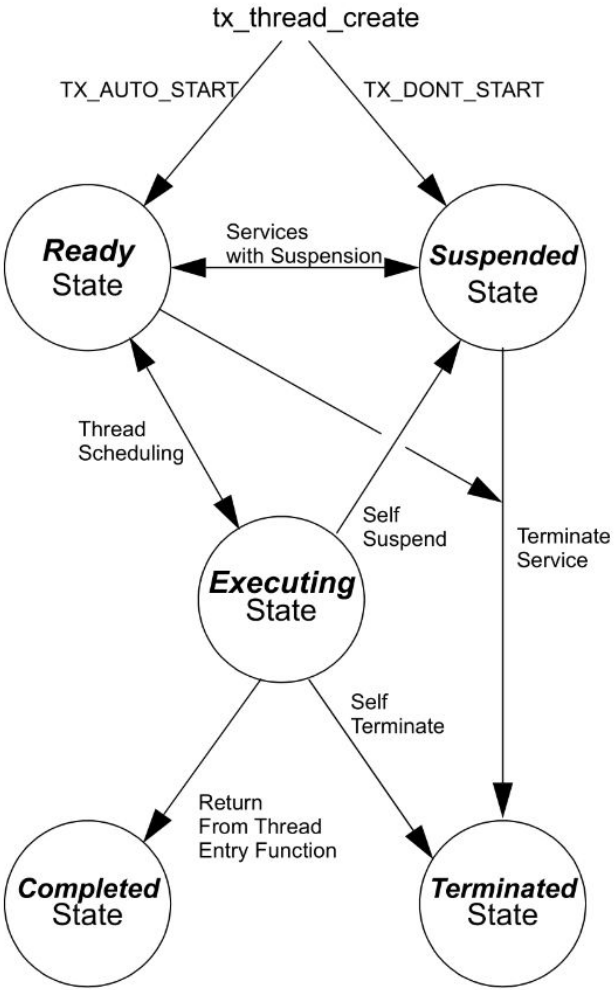
Threads with priority **9–31** ➜ Cannot preempt.

sea|me

software engineering in automotive and mobility ecosystems

preempt_threshold	Highest priority level (0 through (TX_MAX_PRIORITIES-1)) of disabled preemption. Only priorities higher than this level are allowed to preempt this thread. This value must be less than or equal to the specified priority. A value equal to the thread priority disables preemption-threshold.
time_slice	Number of timer-ticks this thread is allowed to run before other ready threads of the same priority are given a chance to run. Note that using preemption-threshold disables time-slicing. Legal time-slice values range from 1 to 0xFFFFFFFF (inclusive). A value of **TX_NO_TIME_SLICE** (a value of 0) disables time-slicing of this thread.
	Using time-slicing results in a slight amount of system overhead. Since time-slicing is only useful in cases where multiple threads share the same priority, threads having a unique priority should not be assigned a time-slice.
auto_start	Specifies whether the thread starts immediately or is placed in a suspended state. Legal options are **TX_AUTO_START** (0x01) and **TX_DONT_START** (0x00). If TX_DONT_START is specified, the application must later call tx_thread_resume in order for the thread to run.

sea|me
software engineering in automotive and mobility ecosystems



```
TX_THREAD      my_thread;
UINT           status;


/* Delete an application thread whose control block is
   "my_thread". Assume that the thread has already been
   created with a call to tx_thread_create.  */
status =  tx_thread_delete(&my_thread);


/* If status equals TX_SUCCESS, the application thread is
   deleted.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_THREAD       my_thread;
UINT            my_old_priority;
UINT            status;


/* Change the thread represented by "my_thread" to priority
   0.  */
status =  tx_thread_priority_change(&my_thread,
                                    0, &my_old_priority);


/* If status equals TX_SUCCESS, the application thread is
   now at the highest priority level in the system.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
UINT tx_thread_priority_change(TX_THREAD *thread_ptr,
                               UINT new_priority, UINT *old_priority)
```

Description

This service changes the priority of the specified thread. Valid priorities range from 0 through (TX_MAX_PRIORITES-1), where 0 represents the highest priority level.

sea|me

software engineering in automotive and mobility ecosystems

Round-robin Scheduling

- ThreadX supports round-robin scheduling for threads with the same priority.
- Is done through cooperative calls to `tx_thread_relinquish`, which gives other ready threads a chance to execute before the calling thread runs again.

```
VOID  tx_thread_relinquish(VOID)
```

Description

This service relinquishes processor control to other ready-to-run threads at the same or higher priority.

sea|me
software engineering in automotive and mobility ecosystems

```
ULONG run_counter_1 =  0;
ULONG run_counter_2 =  0;

/* Example of two threads relinquishing control to
   each other in an infinite loop. Assume that
   both of these threads are ready and have the same
   priority. The run counters will always stay within one
   of each other.  */

VOID  my_first_thread(ULONG thread_input)
{

    /* Endless loop of relinquish.  */
    while(1)
    {

        /* Increment the run counter.  */
        run_counter_1++;

        /* Relinquish control to other thread.  */
        tx_thread_relinquish();
    }
}
```

```
VOID  my_second_thread(ULONG thread_input)
{

    /* Endless loop of relinquish.  */
    while(1)
    {

        /* Increment the run counter.  */
        run_counter_2++;

        /* Relinquish control to other thread.  */
        tx_thread_relinquish();
    }
}
```

sea|me

software engineering in automotive
and mobility ecosystems

Tx_thread_suspend vs tx_thread_sleep

Feature	tx_thread_suspend	tx_thread_sleep
Suspension Type	Indefinite	Fixed time (ticks)
Resume Trigger	Another thread or ISR calls	Automatically after delay expires
Use Case	Event-driven waiting, synchronization	Time delays, periodic execution
CPU Usage	None while suspended	None while sleeping

sea|me
software engineering in automotive and mobility ecosystems

Ticks in Real-Time Systems

- Ticks represent small time intervals used for scheduling and time-based operations.
- With a 160 MHz processor and a 1 ms tick rate, there would be 160,000 ticks per second (1 tick every 6.25 nanoseconds).
- Ticks are used to manage task scheduling, delays, and timers.

```
TX_THREAD          my_thread;
UINT               status;


/* Suspend the thread represented by "my_thread".  */
status =  tx_thread_suspend(&my_thread);


/* If status equals TX_SUCCESS, the application thread is
   unconditionally suspended.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
UINT status;


/* Make the calling thread sleep for 100
   timer-ticks.  */
status =  tx_thread_sleep(100);


/* If status equals TX_SUCCESS, the currently running
   application thread slept for the specified number of
   timer-ticks.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

Tx_thread_resume vs tx_thread_wait_abort

Function	What it does	When to use
`tx_thread_resume()`	Resumes a thread that was explicitly suspended	When you want to restart a suspended thread
`tx_thread_wait_abort()`	Forces a thread out of a wait state (on a resource or event)	When you need to cancel a thread's wait, e.g., timeout or shutdown

sea|me
software engineering in automotive and mobility ecosystems

```
TX_THREAD         my_thread;
UINT              status;


/* Resume the thread represented by "my_thread".  */
status =  tx_thread_resume(&my_thread);


/* If status equals TX_SUCCESS, the application thread is
   now ready to execute.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

```
UINT tx_thread_wait_abort(TX_THREAD *thread_ptr)
```

Description

This service aborts sleep or any other object suspension of the specified thread. If the wait is aborted, a TX_WAIT_ABORTED value is returned from the service that the thread was waiting on.

```
TX_THREAD       my_thread;
UINT            status;


/* Abort the suspension condition of "my_thread." */
status =  tx_thread_wait_abort(&my_thread);


/* If status equals TX_SUCCESS, the thread is now ready
   again, with a return value showing its suspension
   was aborted (TX_WAIT_ABORTED).  */
```

sea|me
software engineering in automotive and mobility ecosystems

```
TX_THREAD               my_thread;

/* Reset the previously created thread "my_thread."  */
status =  tx_thread_reset(&my_thread);

/* If status is TX_SUCCESS the thread is reset.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_THREAD         my_thread;
UINT              status;


/* Terminate the thread represented by "my_thread".  */
status =  tx_thread_terminate(&my_thread);


/* If status equals TX_SUCCESS, the thread is terminated
   and cannot execute again until it is reset.  */
```

sea|me

software engineering in automotive and mobility ecosystems

Adjusting Thread Time-Slice

- **Time-slice control** determines how long a thread runs before being preempted by another thread of equal or higher priority.

- `tx_thread_time_slice_change` function allows setting a specific time slice for a thread, ensuring fair CPU time allocation.

- The function returns **TX_SUCCESS** if the time slice was successfully updated.

sea|me
software engineering in automotive and mobility ecosystems

Example:

- **Thread 1 (Engine Control)**: This thread handles real-time engine management, adjusting fuel injectors and engine timing. It may require a **larger time slice** (50 timer ticks) to ensure it runs continuously and adjusts parameters without interruption.

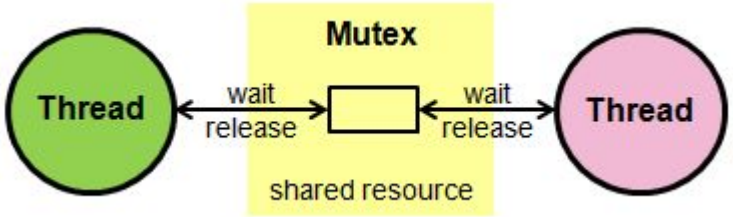
- **Thread 2 (Infotainment System)**: The infotainment system, such as music or navigation, requires less processing time but still needs to run regularly. It can have a **smaller time slice** (20 timer ticks) to ensure the screen updates, without blocking critical tasks like engine control.

```
TX_THREAD         my_thread;
ULONG             my_old_time_slice;
UINT              status;

/* Change the time-slice of the thread associated with
   "my_thread" to 20. This will mean that "my_thread"
   can only run for 20 timer-ticks consecutively before
   other threads of equal or higher priority get a chance
   to run.  */
status =  tx_thread_time_slice_change(&my_thread, 20,
                                      &my_old_time_slice);

/* If status equals TX_SUCCESS, the thread's time-slice
   has been changed to 20 and the previous time-slice is
   in "my_old_time_slice."  */
```

sea|me

software engineering in automotive and mobility ecosystems

MUTEX



```
UINT tx_mutex_create(TX_MUTEX *mutex_ptr,
                     CHAR *name_ptr, UINT priority_inherit)
```

Description

This service creates a mutex for inter-thread mutual exclusion for resource protection.

Parameters

mutex_ptr	Pointer to a mutex control block.
name_ptr	Pointer to the name of the mutex.
priority_inherit	Specifies whether or not this mutex supports priority inheritance. If this value is TX_INHERIT, then priority inheritance is supported. However, if TX_NO_INHERIT is specified, priority inheritance is not supported by this mutex.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_MUTEX      my_mutex;
UINT          status;


/* Create a mutex to provide protection over a
   common resource.  */
status =  tx_mutex_create(&my_mutex,"my_mutex_name",
                          TX_NO_INHERIT);

/* If status equals TX_SUCCESS, my_mutex is ready for
   use.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
UINT tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option)
```

Description

This service attempts to obtain exclusive ownership of the specified mutex. If the calling thread already owns the mutex, an internal counter is incremented and a successful status is returned.

If the mutex is owned by another thread and this thread is higher priority and priority inheritance was specified at mutex create, the lower priority thread's priority will be temporarily raised to that of the calling thread.

The priority of the lower priority thread owning a mutex with priority-inheritance should never be modified by an external thread during mutex ownership.

```
TX_MUTEX      my_mutex;
UINT          status;


/* Release ownership of "my_mutex." */
status =  tx_mutex_put(&my_mutex);


/* If status equals TX_SUCCESS, the mutex ownership
   count has been decremented and if zero, released. */
```

sea|me

software engineering in automotive and mobility ecosystems

Example :

```
tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);      // Lock
shared_counter++;                              // Modify shared resource
tx_mutex_put(&my_mutex);                       // Unlock
```

```
TX_MUTEX      my_mutex;
UINT          status;


/* Delete a mutex. Assume that the mutex
   has already been created.  */
status =  tx_mutex_delete(&my_mutex);


/* If status equals TX_SUCCESS, the mutex is
   deleted.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

`tx_mutex_prioritize(&my_mutex)`

- Ensures that when the mutex becomes available, the thread with the highest priority will be granted ownership of it.
- Useful in real-time operating systems where thread priorities are important to ensure that critical tasks are handled first.

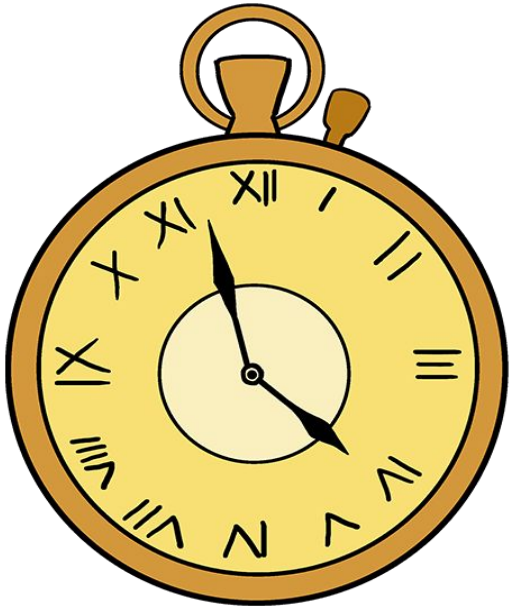
- If the status of the tx_mutex_prioritize function equals **TX_SUCCESS**, it means the highest priority thread is now at the front of the waiting list. When the mutex is unlocked using tx_mutex_put(), this thread will get the mutex and resume execution. Essentially, the thread that was prioritized will now execute.

sea|me
software engineering in automotive and mobility ecosystems

```
TX_MUTEX      my_mutex;
UINT          status;


/* Ensure that the highest priority thread will receive
   ownership of the mutex when it becomes available.  */
status =  tx_mutex_prioritize(&my_mutex);


/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_mutex_put call that releases ownership of the
   mutex will give ownership to this thread and wake it
   up. */
```

sea|me
software engineering in automotive and mobility ecosystems

TIMER



sea|me

software engineering in automotive
and mobility ecosystems

Real-Time Application Timers in ThreadX

- Real-time embedded applications require quick responses to events and periodic activities at specified intervals.
- **ThreadX timers** allow applications to execute C functions at specific time intervals.
- **One-shot timers** expire once, while **periodic timers** repeat at set intervals.
- Each timer is a public resource with no constraints on usage.

sea|me
software engineering in automotive and mobility ecosystems

```
UINT tx_timer_create(TX_TIMER *timer_ptr, CHAR *name_ptr,
                     VOID (*expiration_function)(ULONG),
                     ULONG expiration_input, ULONG initial_ticks,
                     ULONG reschedule_ticks, UINT auto_activate)
```

This function sets up an application timer that calls a function after a specified time, potentially repeatedly.

sea|me
software engineering in automotive and mobility ecosystems

```
TX_TIMER          my_timer;
UINT              status;


/* Create an application timer that executes
   "my_timer_function" after 100 ticks initially and then
   after every 25 ticks. This timer is specified to start
   immediately!  */
status =  tx_timer_create(&my_timer,"my_timer_name",
                          my_timer_function, 0x1234, 100, 25,
                          TX_AUTO_ACTIVATE);


/* If status equals TX_SUCCESS, my_timer_function will
   be called 100 timer ticks later and then called every
   25 timer ticks. Note that the value 0x1234 is passed to
   my_timer_function every time it is called.  */
```

sea|me

software engineering in automotive and mobility ecosystems

1. **timer_ptr** (TX_TIMER*):
   - Pointer to a timer object that will hold the created timer.
2. **name_ptr** (CHAR*):
   - A string representing the name of the timer.
3. **expiration_function** (VOID (*)(ULONG)):
   - A function that is called when the timer expires, taking a ULONG input.
4. **expiration_input** (ULONG):
   - An input value passed to the expiration function when the timer expires.
5. **initial_ticks** (ULONG):
   - The number of ticks before the timer first expires.
6. **reschedule_ticks** (ULONG):
   - The number of ticks between subsequent timer expirations (for periodic timers).
7. **auto_activate** (UINT):
   A flag that determines whether the timer should start automatically after creation.

If auto_activate is not set we can manually activate after

```
TX_TIMER          my_timer;
UINT              status;


/* Activate an application timer. Assume that the
   application timer has already been created.  */
status =  tx_timer_activate(&my_timer);


/* If status equals TX_SUCCESS, the application timer is
   now active.  */
```

sea|me
software engineering in automotive and mobility ecosystems

```
UINT tx_timer_deactivate(TX_TIMER *timer_ptr)
```

sea|me

software engineering in automotive
and mobility ecosystems

VOID **tx_time_set**(ULONG **new_time**)

Description

This service sets the internal system clock to the specified value. Each timer-tick increases the internal system clock by one.

Let's say you have a system with 3 devices: **Device A**, **Device B**, and **Device C**. The central controller sends a synchronization signal at **time 1000**.

- **Before synchronization**, the devices have their individual clocks like this:
  - Device A: Clock = 300
  - Device B: Clock = 500
  - Device C: Clock = 1200
- When each device receives the signal, it calls `tx_time_set(1000)` to set its clock to 1000:
  - Device A: Clock = 1000
    Device B: Clock = 1000
  - Device C: Clock = 1000
- After synchronization, each device starts ticking from 1000, and all timers are now aligned to perform tasks based on the synchronized time.

The value of the system clock (e.g., **10000** or **300**) represents the **number of timer ticks** that have passed from the time the clock was set or started. A timer "tick" is just an increment in time, typically corresponding to a specific interval or time period.

sea|me
software engineering in automotive and mobility ecosystems

Let’s say the system timer is set to tick every **1 millisecond**. The system clock will count the number of milliseconds that have passed. Here’s how the clock values relate to real time:

- **System clock = 10000**:
  - The system has been running for **10 seconds**.
  - 10,000 timer ticks have occurred.
  - If you set the clock to **10000** (using `tx_time_set(10000)`), you are telling the system, "Start counting from 10,000 timer ticks."
- **System clock = 300**:
  - The system has been running for **0.3 seconds**.
  - 300 timer ticks have occurred.
  - If you set the clock to **300**, the system would start from that point in time, with the next tick occurring at the next 301st tick.

sea|me
software engineering in automotive and mobility ecosystems

```
UINT tx_timer_change(TX_TIMER *timer_ptr,
                     ULONG initial_ticks, ULONG reschedule_ticks)
```

Description

This service changes the expiration characteristics of the specified application timer. The timer must be deactivated prior to calling this service.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_TIMER          my_timer;
UINT              status;


/* Change a previously created and now deactivated timer
   to expire every 50 timer ticks, including the initial
   expiration.  */
status =  tx_timer_change(&my_timer,50, 50);


/* If status equals TX_SUCCESS, the specified timer is
   changed to expire every 50 ticks.  */

/* Activate the specified timer to get it started again.  */
status =  tx_timer_activate(&my_timer);
```

sea|me

software engineering in automotive and mobility ecosystems

```
ULONG tx_time_get(VOID)
```

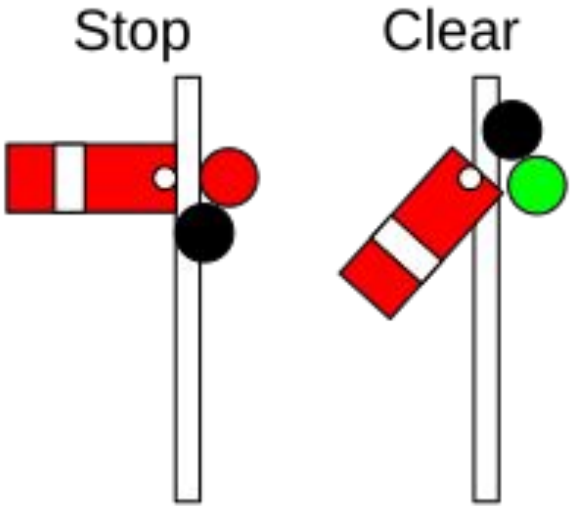
Description

This service returns the contents of the internal system clock. Each timer-tick increases the internal system clock by one. The system clock is set to zero during initialization and can be changed to a specific value by the service ***tx_time_set***.

sea|me

software engineering in automotive and mobility ecosystems

SEMAPHORE



ThreadX 32-bit Counting Semaphores

- A semaphore is like a **counter** or **token system** for coordinating access to shared resources or events.
- Range: 0 to 4,294,967,295.

- **Operations**:

  - **tx_semaphore_get**: Decreases semaphore by 1. Fails if semaphore is 0.

  - **tx_semaphore_put**: Increases semaphore by 1.

sea|me
software engineering in automotive and mobility ecosystems

Purpose of `tx_semaphore_create`

- **Creates a counting semaphore** for **inter-thread synchronization**.
- It allocates the internal control structure needed by ThreadX.

What Happens Internally

1. **ThreadX** initializes the semaphore object.
2. The semaphore's **count** is set to `initial_count`.
3. **ThreadX** sets the internal suspension list for threads that will wait on the semaphore.
4. After creation, threads can call:
    - `tx_semaphore_get()` → **Decrement** count (wait for token)

      `tx_semaphore_put()` → **Increment** count (release a token)

sea|me
software engineering in automotive and mobility ecosystems

```
UINT tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr,
                         CHAR *name_ptr, ULONG initial_count)

        TX_SEMAPHORE        my_semaphore;
        UINT                status;


        /* Create a counting semaphore whose initial value is 1.
           This is typically the technique used to make a binary
           semaphore. Binary semaphores are used to provide
           protection over a common resource.  */
        status =  tx_semaphore_create(&my_semaphore,
                                      "my_semaphore_name", 1);


        /* If status equals TX_SUCCESS, my_semaphore is ready for
           use.  */
```

sea|me

software engineering in automotive and mobility ecosystems

Car Parking System with Semaphores

- **Scenario**: Managing car entry and parking in a smart parking garage.
- **Key Resources**: Parking slots, sensors, gate control.
- **Semaphore**: Used to track available parking slots.

Flow:

1. **Semaphore Creation**: Initializes semaphore with the number of available slots.
2. **Car Entry**: Car checks semaphore; if slots are free, it parks (decrements semaphore).
3. **Car Exit**: Car exits, semaphore is incremented, slot is free for another car.
4. **Full Parking**: If semaphore count is 0, new cars wait until slots become available.

Benefits:

- **Controls access** to parking slots.
- **Synchronizes** car entry and exit.
- **Prevents overflow** when the garage is full.

```
TX_SEMAPHORE my_semaphore;
UINT         status;

/* Get a semaphore instance from the semaphore
   "my_semaphore." If the semaphore count is zero,
   suspend until an instance becomes available.
   Note that this suspension is only possible from
   application threads.  */
status =  tx_semaphore_get(&my_semaphore, TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the thread has obtained
   an instance of the semaphore.  */
```

sea!me

software engineering in automotive and mobility ecosystems

```
UINT tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr)
```

Description

This service puts an instance into the specified counting semaphore, which in reality increments the counting semaphore by one.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_SEMAPHORE        my_semaphore;
UINT                status;


/* Increment the counting semaphore "my_semaphore." */
status =  tx_semaphore_put(&my_semaphore);


/* If status equals TX_SUCCESS, the semaphore count has
   been incremented. Of course, if a thread was waiting,
   it was given the semaphore instance and resumed.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_SEMAPHORE        my_semaphore;
UINT                status;


/* Delete counting semaphore. Assume that the counting
   semaphore has already been created.  */
status =  tx_semaphore_delete(&my_semaphore);

/* If status equals TX_SUCCESS, the counting semaphore is
   deleted.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

```
UINT  tx_semaphore_ceiling_put(TX_SEMAPHORE *semaphore_ptr,
        ULONG ceiling);
```

Description

This service puts an instance into the specified counting semaphore, which in reality increments the counting semaphore by one. If the counting semaphore's current value is greater than or equal to the specified ceiling, the instance will not be put and a TX_CEILING_EXCEEDED error will be returned.

```
TX_SEMAPHORE     my_semaphore;


/* Increment the counting semaphore "my_semaphore" but make sure
   that it never exceeds 7 as specified in the call.  */
status =  tx_semaphore_ceiling_put(&my_semaphore, 7);

/* If status is TX_SUCCESS the semaphore count has been
   incremented.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

```
UINT  tx_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr,
        VOID (*semaphore_put_notify)(TX_SEMAPHORE *));
```

Description

This service registers a notification callback function that is called whenever the specified semaphore is put. The processing of the notification callback is defined by the application.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_SEMAPHORE     my_semaphore;

/* Register the "my_semaphore_put_notify" function for monitoring
   the put operations on the semaphore "my_semaphore."  */
status =  tx_semaphore_put_notify(&my_semaphore,
                  my_semaphore_put_notify);

/* If status is TX_SUCCESS the semaphore put notification function
   was successfully registered.  */

void my_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr)
{
    /* The semaphore was just put!  */
}
```

sea|me
software engineering in automotive and mobility ecosystems

```
TX_SEMAPHORE my_semaphore;
UINT         status;


/* Ensure that the highest priority thread will receive
   the next instance of this semaphore.  */
status =  tx_semaphore_prioritize(&my_semaphore);


/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_semaphore_put call made to this semaphore will
   wake up this thread. */
```

sea|me

software engineering in automotive and mobility ecosystems

Semaphore vs Mutex

- **Semaphore**:
  - **Purpose**: Manages multiple resources .
  - **Count**: Allows multiple threads to access the resource (based on the counter).
  - **No Ownership**: Any thread can signal or wait.
  - **Risk of Race Conditions**: Proper usage is important to avoid race conditions, especially when modifying shared resources.
- **Mutex**:
  - **Purpose**: Protects critical sections (e.g., shared data).
  - **Ownership**: Only the thread that locks the mutex can unlock it.
  - **Single Access**: Only one thread can access the resource at a time.

Key Notes:

- **Semaphore**: Ideal for limiting access to a fixed number of resources.
- **Mutex**: Ensures exclusive access to critical sections, preventing race conditions.

QUEUE

Queues in ThreadX?

- **Creation**: Can be created during initialization or at runtime by application threads. There is no limit to the number of message queues.
- **Definition**: Allow threads to **send** and **receive** messages safely.
- **Function**: Used for inter-thread communication, where one or more messages can reside in the queue.
- **How It Works**:
  - One thread **sends** a message into the queue.
  - Another thread **receives** it later (FIFO - First In, First Out).

sea|me
software engineering in automotive and mobility ecosystems

Why Use Queues?

- **Purpose**: To pass data or commands between threads without directly sharing memory (avoiding race conditions).

- **Common Use Cases**:
  - **Producer** thread generates data (e.g., sensor readings) and sends it to a **consumer** thread for processing.
  - **ISR** (Interrupt Service Routine) sends short messages to threads for deferred processing.
  - Threads exchange commands/results asynchronously.

sea|me
software engineering in automotive and mobility ecosystems

```
UINT tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr,
                     UINT message_size,
                     VOID *queue_start, ULONG queue_size)
```

Description

This service creates a message queue that is typically used for inter-thread communication. The total number of messages is calculated from the specified message size and the total number of bytes in the queue.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_QUEUE      my_queue;
UINT          status;

/* Create a message queue whose total size is 2000 bytes
   starting at address 0x300000. Each message in this
   queue is defined to be 4 32-bit words long.  */
status =  tx_queue_create(&my_queue, "my_queue_name",
                          4, (VOID *) 0x300000, 2000);

/* If status equals TX_SUCCESS, my_queue contains room
   for storing 125 messages (2000 bytes/ 16 bytes per
   message).  */
```

sea|me

software engineering in automotive and mobility ecosystems

queue_ptr	Pointer to a message queue control block.
name_ptr	Pointer to the name of the message queue.
message_size	Specifies the size of each message in the queue. Message sizes range from 1 32-bit word to 16 32-bit words. Valid message size options are numerical values from 1 through 16, inclusive.
queue_start	Starting address of the message queue. The starting address must be aligned to the size of the ULONG data type.
queue_size	Total number of bytes available for the message queue.

sea|me
software engineering in automotive and mobility ecosystems

Queue send

Function	Position in Queue	Purpose	Use Case
`tx_queue_send()`	End of the queue	Sends message in FIFO order	Normal message processing (FIFO)
`tx_queue_front_send()`	Front of the queue	Sends message with higher priority	Prioritizing messages (time-sensitive)

sea|me
software engineering in automotive and mobility ecosystems

```
TX_QUEUE      my_queue;
UINT          status;
ULONG         my_message[4];


/* Send a message to the front of "my_queue." Return
   immediately, regardless of success. This wait
   option is used for calls from initialization, timers,
   and ISRs.  */
status =  tx_queue_front_send(&my_queue, my_message,
                              TX_NO_WAIT);


/* If status equals TX_SUCCESS, the message is at the front
   of the specified queue. */
```

sea|me

software engineering in automotive and mobility ecosystems

Prototype

```
UINT tx_queue_send(TX_QUEUE *queue_ptr,
                   VOID *source_ptr, ULONG wait_option)
```

Description

This service sends a message to the specified message queue. The sent message is **copied** to the queue from the memory area specified by the source pointer.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_QUEUE        my_queue;
UINT            status;
ULONG           my_message[4];


/* Send a message to "my_queue." Return immediately,
   regardless of success. This wait option is used for
   calls from initialization, timers, and ISRs.  */
status =  tx_queue_send(&my_queue, my_message, TX_NO_WAIT);


/* If status equals TX_SUCCESS, the message is in the
   queue.  */
```

sea|me
software engineering in automotive
and mobility ecosystems

```
UINT tx_queue_receive(TX_QUEUE *queue_ptr,
                      VOID *destination_ptr, ULONG wait_option)
```

Description

This service retrieves a message from the specified message queue. The retrieved message is **copied** from the queue into the memory area specified by the destination pointer. That message is then removed from the queue.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_QUEUE         my_queue;
UINT             status;
ULONG            my_message[4];


/* Retrieve a message from "my_queue." If the queue is
   empty, suspend until a message is present. Note that
   this suspension is only possible from application
   threads.  */
status =  tx_queue_receive(&my_queue, my_message,
                           TX_WAIT_FOREVER);


/* If status equals TX_SUCCESS, the message is in
   "my_message."  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_QUEUE      my_queue;
UINT          status;


/* Delete entire message queue. Assume that the queue
   has already been created with a call to
   tx_queue_create.  */
status =  tx_queue_delete(&my_queue);


/* If status equals TX_SUCCESS, the message queue is
   deleted.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

```
UINT tx_queue_flush(TX_QUEUE *queue_ptr)
```

Description

This service deletes all messages stored in the specified message queue. If the queue is full, messages of all suspended threads are discarded. Each suspended thread is then resumed with a return status that indicates the message send was successful. If the queue is empty, this service does nothing.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_QUEUE      my_queue;
UINT          status;


/* Flush out all pending messages in the specified message
   queue. Assume that the queue has already been created
   with a call to tx_queue_create.  */
status =  tx_queue_flush(&my_queue);


/* If status equals TX_SUCCESS, the message queue is
   empty.  */
```

sea|me

software engineering in automotive
and mobility ecosystems

Prototype

```
UINT tx_queue_prioritize(TX_QUEUE *queue_ptr)
```

Description

This service places the highest priority thread suspended for a message (or to place a message) on this queue at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

sea|me
software engineering in automotive and mobility ecosystems

```
TX_QUEUE      my_queue;
UINT          status;


/* Ensure that the highest priority thread will receive
   the next message placed on this queue.  */
status =  tx_queue_prioritize(&my_queue);


/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_queue_send or tx_queue_front_send call made
   to this queue will wake up this thread. */
```

sea|me
software engineering in automotive
and mobility ecosystems

```
UINT  tx_queue_send_notify(TX_QUEUE *queue_ptr,
        VOID (*queue_send_notify)(TX_QUEUE *));
```

Purpose: Sends a message to a queue and notifies a waiting thread that the message is available.

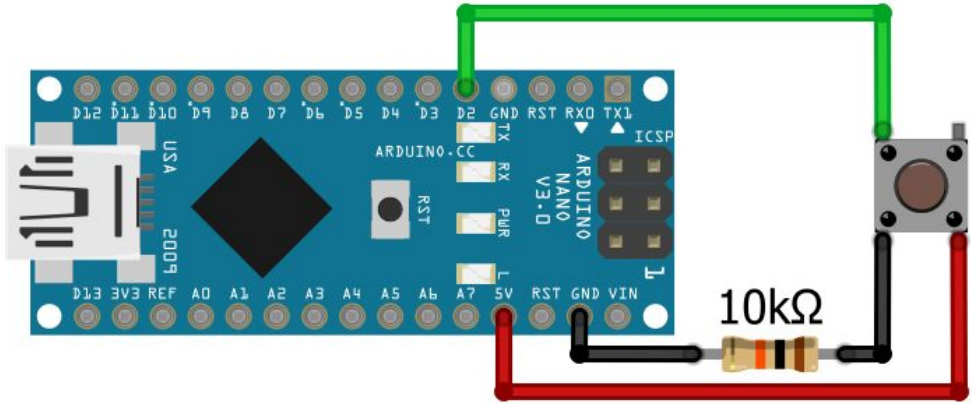
When to Use: Use this function when you want to send a message to a queue and wake up a thread that may be waiting for a message in that queue.

Key Feature: It combines sending a message to the queue and notifying the waiting thread in a single operation.

```
TX_QUEUE         my_queue;


/* Register the "my_queue_send_notify" function for monitoring
   messages sent to the queue "my_queue."  */
status =  tx_queue_send_notify(&my_queue, my_queue_send_notify);


/* If status is TX_SUCCESS the queue send notification function was
   successfully registered.  */
void my_queue_send_notify(TX_QUEUE *queue_ptr)
{

    /* A message was just sent to this queue!  */

}
```

sea|me
software engineering in automotive
and mobility ecosystems

INTERRUPT



driva|pi

sea|me

software engineering in automotive and mobility ecosystems

```
UINT my_old_posture;

/* Lockout interrupts */
my_old_posture =  tx_interrupt_control(TX_INT_DISABLE);

/* Perform critical operations that need interrupts
   locked-out....  */

/* Restore previous interrupt lockout posture.  */
tx_interrupt_control(my_old_posture);
```

sea|me

software engineering in automotive
and mobility ecosystems

tx_interrupt_control

Key Functions:

- **Disable Interrupts**:
  Temporarily disables interrupts to protect critical sections.

  ```
  tx_interrupt_control(TX_INT_DISABLE);
  ```

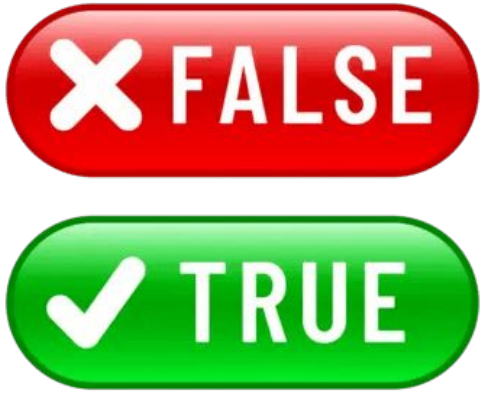
- **Enable Interrupts**:

  Re-enables interrupts after critical sections.

  ```
  tx_interrupt_control(TX_INT_ENABLE);
  ```

sea|me
software engineering in automotive
and mobility ecosystems

FLAGS



```
TX_EVENT_FLAGS_GROUP    my_event_group;
UINT                    status;


/* Create an event flags group.  */
status =  tx_event_flags_create(&my_event_group,
                    "my_event_group_name");


/* If status equals TX_SUCCESS, my_event_group is ready
   for get and set services. */
```

sea|me

software engineering in automotive and mobility ecosystems

```
UINT tx_event_flags_create(TX_EVENT_FLAGS_GROUP *group_ptr,
                           CHAR *name_ptr)
```

Description

This service creates a group of 32 event flags. All 32 event flags in the group are initialized to zero. Each event flag is represented by a single bit.

Parameters

group_ptr	Pointer to an event flags group control block.
name_ptr	Pointer to the name of the event flags group.

Return Values

TX_SUCCESS	(0x00)	Successful event group creation.
TX_GROUP_ERROR	(0x06)	Invalid event group pointer. Either the pointer is NULL or the event group is already created.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

```
UINT tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr,
                        ULONG requested_flags, UINT get_option,
                        ULONG *actual_flags_ptr, ULONG wait_option)
```

Description

This service retrieves event flags from the specified event flags group. Each event flags group contains 32 event flags. Each flag is represented by a single bit. This service can retrieve a variety of event flag combinations, as selected by the input parameters.

Parameters

group_ptr — Pointer to a previously created event flags group.

requested_flags — 32-bit unsigned variable that represents the requested event flags.

get_option — Specifies whether all or any of the requested event flags are required. The following are valid selections:

TX_AND	(0x02)
TX_AND_CLEAR	(0x03)
TX_OR	(0x00)
TX_OR_CLEAR	(0x01)

Selecting TX_AND or TX_AND_CLEAR specifies that all event flags must be present in the group. Selecting TX_OR or TX_OR_CLEAR specifies that any event flag is satisfactory. Event flags that satisfy the request are cleared (set to zero) if TX_AND_CLEAR or TX_OR_CLEAR are specified.

actual_flags_ptr — Pointer to destination of where the retrieved

Prototype

```
UINT tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr,
                        ULONG  flags_to_set,UINT  set_option)
```

sea|me

software engineering in automotive
and mobility ecosystems

`TX_EVENT_FLAGS_GROUP *group_ptr`:

- This is a pointer to the **event flags group** in which the flags will be set.
- An event flags group is a collection of flags (typically stored in a bitmask), which threads can manipulate to signal and synchronize with each other.

`ULONG flags_to_set`:

- This is the **bitmask** representing the flags you want to set.
- Each bit in the bitmask corresponds to an individual flag in the event flags group. When you set a bit, it means you are setting the corresponding event flag.
- For example, if `flags_to_set` is `0x01`, it means you are setting the first flag.

`UINT set_option`:

- This parameter defines how the flags are set.
- The typical options might include:
  - **TX_OR**: Set the flags specified by `flags_to_set` in the group.
  - **TX_AND**: Set the flags only if the current flags match a specified condition.
  - **TX_AND_CLEAR**: Set flags and also clear them from the group.

sea|me
software engineering in automotive and mobility ecosystems

TX_OR:

```
Current flags:     00000000 00000000 00000000 00101000
Flags to set:      00000000 00000000 00000000 00100100
----------------------------
TX_OR result:      00000000 00000000 00000000 00101100
```

TX_AND operation:

```
Current flags:     00000000 00000000 00000000 00101000
Flags to set:      00000000 00000000 00000000 00100100
----------------------------
TX_OR result:      00000000 00000000 00000000 00101100
```

TX_AND_CLEAR operation:

```
Current flags:     00000000 00000000 00000000 00101000
Flags to set:      00000000 00000000 00000000 00100100
----------------------------
TX_AND_CLEAR result:
                   00000000 00000000 00000000 00000000
```

sea|me
software engineering in automotive and mobility ecosystems

bitmask

In a **bitmask**, each **bit** represents a different flag. For example, a 32-bit bitmask (in a 32-bit system) has 32 bits, where each bit can represent a flag (from `flag 0` to `flag 31`).

Example: Set Flags 5, 9, and 22

1. **Creating a Bitmask to Set Flags 5, 9, and 22**:
   - You need to **set bits 5, 9, and 22** to `1`, leaving the rest as `0`.
2. Here's how to create the bitmask:
   - `1 << 5` will shift the number `1` to position 5 in the bitmask, which gives `0x20` or `00000000000100000000000000000000`.
   - `1 << 9` will shift the number `1` to position 9, which gives `0x200` or `00000000000000100000000000000000`.
   - `1 << 22` will shift the number `1` to position 22, which gives `0x400000` or `00000000000000000001000000000000`.
   - ULONG flags_to_set = (1 << 5) | (1 << 9) | (1 << 22);
     1. 00000001000100010010000000000000

sea|me
software engineering in automotive and mobility ecosystems

```
TX_EVENT_FLAGS_GROUP    my_event_flags_group;
ULONG                   actual_events;
UINT                    status;


/* Request that event flags 0, 4, and 8 are all set. Also,
   if they are set they should be cleared. If the event
   flags are not set, this service suspends for a maximum of
   20 timer-ticks.  */
status =  tx_event_flags_get(&my_event_flags_group, 0x111,
                        TX_AND_CLEAR, &actual_events, 20);


/* If status equals TX_SUCCESS, actual_events contains the
   actual events obtained.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
UINT  tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr,
        VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));
```

Description

This service registers a notification callback function that is called whenever one or more event flags are set in the specified event flags group. The processing of the notification callback is defined by the application.

sea|me

software engineering in automotive and mobility ecosystems

```
TX_EVENT_FLAGS_GROUP     my_group;

/* Register the "my_event_flags_set_notify" function for monitoring
   event flags set in the event flags group "my_group."  */
status =  tx_event_flags_set_notify(&my_group,
                                    my_event_flags_set_notify);

/* If status is TX_SUCCESS the event flags set notification functio
   was successfully registered.  */

void my_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr)
   /* One or more event flags was set in this group!  */
```

sea|me
software engineering in automotive
and mobility ecosystems

```c
typedef enum {
    FLAG_A = (1 << 0),
    FLAG_B = (1 << 1),
    FLAG_C = (1 << 2)
} t_flag;

int f(int x, t_flag flags)
{
    if (flags & FLAG_A)
        x += x;
    if (flags & FLAG_B)
        x *= x;
    if (flags & FLAG_C)
        x = ~x;
    return x;
}

int main(void) {
    printf("%d\n", f(1234, 0));
    printf("%d\n", f(1234, FLAG_A));
    printf("%d\n", f(1234, FLAG_B | FLAG_C));
    return 0;
}
```

```c
int main(void) {
    printf("%d\n", f(1234, false, false, false));
    printf("%d\n", f(1234, true, false, false));
    printf("%d\n", f(1234, false, true, true));
    return 0;
}
```

sea|me

software engineering in automotive
and mobility ecosystems

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT                 status;

/* Delete event flags group. Assume that the group has
   already been created with a call to
   tx_event_flags_create.  */
status =  tx_event_flags_delete(&my_event_flags_group);

/* If status equals TX_SUCCESS, the event flags group is
   deleted.  */
```

sea|me
software engineering in automotive
and mobility ecosystems

POOLS

sea|me

software engineering in automotive
and mobility ecosystems

```
UINT tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
                          CHAR *name_ptr, ULONG block_size,
                          VOID *pool_start, ULONG pool_size)
```

Description

This service creates a pool of fixed-size memory blocks. The memory area specified is divided into as many fixed-size memory blocks as possible using the formula:

total blocks = (**total bytes**) / (**block size** + sizeof(void *))

*Each memory block contains one pointer of overhead that is invisible to the user and is represented by the "sizeof(void *)" in the preceding formula.*

Parameters

pool_ptr	Pointer to a memory block pool control block.
name_ptr	Pointer to the name of the memory block pool.
block_size	Number of bytes in each memory block.
pool_start	Starting address of the memory block pool. The starting address must be aligned to the size of the ULONG data type.
pool_size	Total number of bytes available for the memory block pool.

A **block pool** is a *fixed-size memory allocator*.

```
TX_BLOCK_POOL   my_pool;
UINT            status;


/* Create a memory pool whose total size is 1000 bytes
   starting at address 0x100000. Each block in this
   pool is defined to be 50 bytes long.  */
status =  tx_block_pool_create(&my_pool, "my_pool_name",
                50, (VOID *) 0x100000, 1000);


/* If status equals TX_SUCCESS, my_pool contains 18
   memory blocks of 50 bytes each. The reason
   there are not 20 blocks in the pool is
   because of the one overhead pointer associated with each
   block.  */
```

(To store information like block availability or block metadata)

sea|me
software engineering in automotive and mobility ecosystems

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;
UINT          status;


/* Allocate a memory block from my_pool. Assume that the
   pool has already been created with a call to
   tx_block_pool_create.  */
status = tx_block_allocate(&my_pool, (VOID **) &memory_ptr,
                                     TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated block of memory.  */
```

sea|me

software engineering in automotive and mobility ecosystems

Why not malloc ?

- `Malloc` is a general-purpose dynamic memory allocator
- **Memory pool** like `tx_block_pool_create` is a more specialized memory management tool for efficiently allocating fixed-size memory blocks, especially in embedded systems or real-time applications.
- The pool can improve performance and reduce fragmentation compared to using `malloc` for multiple small, fixed-size allocations.

sea|me
software engineering in automotive and mobility ecosystems

Memory Pool Creation (Setup Phase):

- The function `tx_block_pool_create()` is used to **set up** the memory pool. It reserves a certain amount of memory and organizes it into fixed-size blocks, but no actual memory is being **used** yet at this point.
- This function simply establishes the pool's structure, defines how many blocks can fit into the pool, and prepares the memory for future allocation. The pool is just a container or a set of available memory slots at this stage.

Memory Allocation (Execution Phase):

- After the pool is created, you need to **allocate** memory from it when you need to actually use it in your program. This is where a function like `tx_block_allocate()` comes into play. This function takes a block from the pool and makes it available for use in your program.

Prototype

```
UINT tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr,
                       ULONG wait_option)
```

Description

This service allocates a fixed-size memory block from the specified memory pool. The actual size of the memory block is determined during memory pool creation.

It is important to ensure application code does not write outside the allocated memory block. If this happens, corruption occurs in an adjacent (usually subsequent) memory block. The results are unpredictable and often fatal!

Parameters

pool_ptr	Pointer to a previously created memory block pool.
block_ptr	Pointer to a destination block pointer. On successful allocation, the address of the allocated memory block is placed where this parameter points.
wait_option	Defines how the service behaves if there are no memory blocks available. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

```
TX_BLOCK_POOL           my_pool;
unsigned char           *memory_ptr;
UINT                    status;


/* Release a memory block back to my_pool. Assume that the
   pool has been created and the memory block has been
   allocated.  */
status =  tx_block_release((VOID *) memory_ptr);


/* If status equals TX_SUCCESS, the block of memory pointed
   to by memory_ptr has been returned to the pool.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_BLOCK_POOL  my_pool;
UINT           status;


    /* Delete entire memory block pool. Assume that the pool
       has already been created with a call to
       tx_block_pool_create.  */
    status =  tx_block_pool_delete(&my_pool);


    /* If status equals TX_SUCCESS, the memory block pool is
       deleted.  */
```

sea|me

software engineering in automotive and mobility ecosystems

A **byte pool** is a *variable-size memory allocator*.

```
TX_BYTE_POOL my_pool;
UINT         status;


/* Create a memory pool whose total size is 2000 bytes
   starting at address 0x500000.  */
status =  tx_byte_pool_create(&my_pool, "my_pool_name",
              (VOID *) 0x500000, 2000);


/* If status equals TX_SUCCESS, my_pool is available for
   allocating memory.  */
```

sea|me
software engineering in automotive
and mobility ecosystems

```
TX_BYTE_POOL my_pool;
unsigned char*memory_ptr;
UINT          status;


/* Allocate a 112 byte memory area from my_pool. Assume
   that the pool has already been created with a call to
   tx_byte_pool_create.  */
status =  tx_byte_allocate(&my_pool, (VOID **) &memory_ptr,
                           112, TX_NO_WAIT);


/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated memory area.  */
```

sea|me

software engineering in automotive and mobility ecosystems

```
TX_BYTE_POOL my_pool;
UINT         status;


/* Delete entire memory pool. Assume that the pool has already
   been created with a call to tx_byte_pool_create.  */
status =  tx_byte_pool_delete(&my_pool);


/* If status equals TX_SUCCESS, memory pool is deleted. */
```

sea|me
software engineering in automotive
and mobility ecosystems

```
TX_BYTE_POOL my_pool;
UINT         status;


/* Ensure that the highest priority thread will receive
   the next free memory from this pool.  */
status =  tx_byte_pool_prioritize(&my_pool);


/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_byte_release call will wake up this thread,
   if there is enough memory to satisfy its request. */
```

sea|me

software engineering in automotive and mobility ecosystems

```
unsigned char    *memory_ptr;
UINT             status;

/* Release a memory back to my_pool. Assume that the memory
   area was previously allocated from my_pool.  */
status =  tx_byte_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the memory pointed to by
   memory_ptr has been returned to the pool.  */
```

sea|me
software engineering in automotive and mobility ecosystems

BLOCK POOL VS BYTE POOL

A **Block Pool** divides a region of memory into **equal-size blocks**, for example:

[########][########][########][########][########]

Block 1 Block 2 Block 3 Block 4 Block 5

When the system needs memory (`tx_block_allocate()`):

- It simply **takes the first free block** from a list.
- No searching, no splitting, no merging.

sea|me
software engineering in automotive and mobility ecosystems

BLOCK POOL VS BYTE POOL

A **Byte Pool** manages a memory area for **variable-size allocations**, kind of like a heap:

When you request, say, 12 bytes:

The system must search the list of free regions to find one large enough.

Then it splits that region (12 bytes allocated + the rest left free).

When you free memory, it may need to merge neighboring free regions to reduce fragmentation.

After several allocations and frees, memory might look like this:

[###][----][#####][--][##########][---]

A free B free C free

sea|me

software engineering in automotive and mobility ecosystems

Feature	Byte Pool	Block Pool
Allocation size	Variable	Fixed
Fragmentation	Possible	None
Allocation speed	Slower	Very fast
Memory efficiency	High initially, can degrade	Consistent
Use case	Dynamic buffers, strings, variable data	Packets, objects, queues
API functions	tx_byte_*	tx_block_*

sea|me
software engineering in automotive and mobility ecosystems

Function	Purpose	Allocates	Notes
🧱 Block Pool			
`tx_block_pool_create()`	Set up **fixed-size** memory pool	—	Simple and deterministic; divides memory into equal blocks
`tx_block_allocate()`	Get one **fixed-size block**	Fixed-size	Very fast; no fragmentation
`tx_block_release()`	Return a block to the pool	—	Constant-time operation; predictable performance
🧩 Byte Pool			
`tx_byte_pool_create()`	Set up **variable-size** memory pool	—	Manages a memory region for variable-size allocations
`tx_byte_allocate()`	Allocate a **requested number of bytes**	Variable-size	Flexible, but may cause fragmentation; slower than block pool
`tx_byte_release()`	Free a previously allocated memory block	—	Returns memory to the pool; can leave fragmented gaps

sea|me
software engineering in automotive and mobility ecosystems

Resources

Microsoft Corporation. *Azure RTOS ThreadX User Guide.* Published February 2020.

sea|me
software engineering in automotive and mobility ecosystems

THANK YOU

driva|pi

sea|me

software engineering in automotive
and mobility ecosystems