ThreadX Kernel API's

NetSilicon

ThreadX in General

- ThreadX is delivered in binary format
- No MMU support
- Resides with Application
- System structures directly visible to Application
- Multilevel Queue Scheduler
  - 32 Thread priorities (0 highest, 31 lowest)
  - Timeslicing requires HW Timer
- Small Footprint (1.7kB to 11.2kB text)
- Unlimited Threads, Queues, Event Flags, …
- Runs completely in Supervisor Mode

NetSilicon

Kernel API's Overview

4.1 Thread Management

4.2 Memory Management

4.3 Semaphores

4.4 Event Flags

4.5 Message Queues

4.6 Timers

4.7 Other Kernel Services

4.1 Thread Management

NetSilicon

ThreadX™ RTOS

- Threads
- Message Queues
- Semaphores
- Event Flags
- Timers
- Memory Management

Under 25 Kbytes Instruction Area

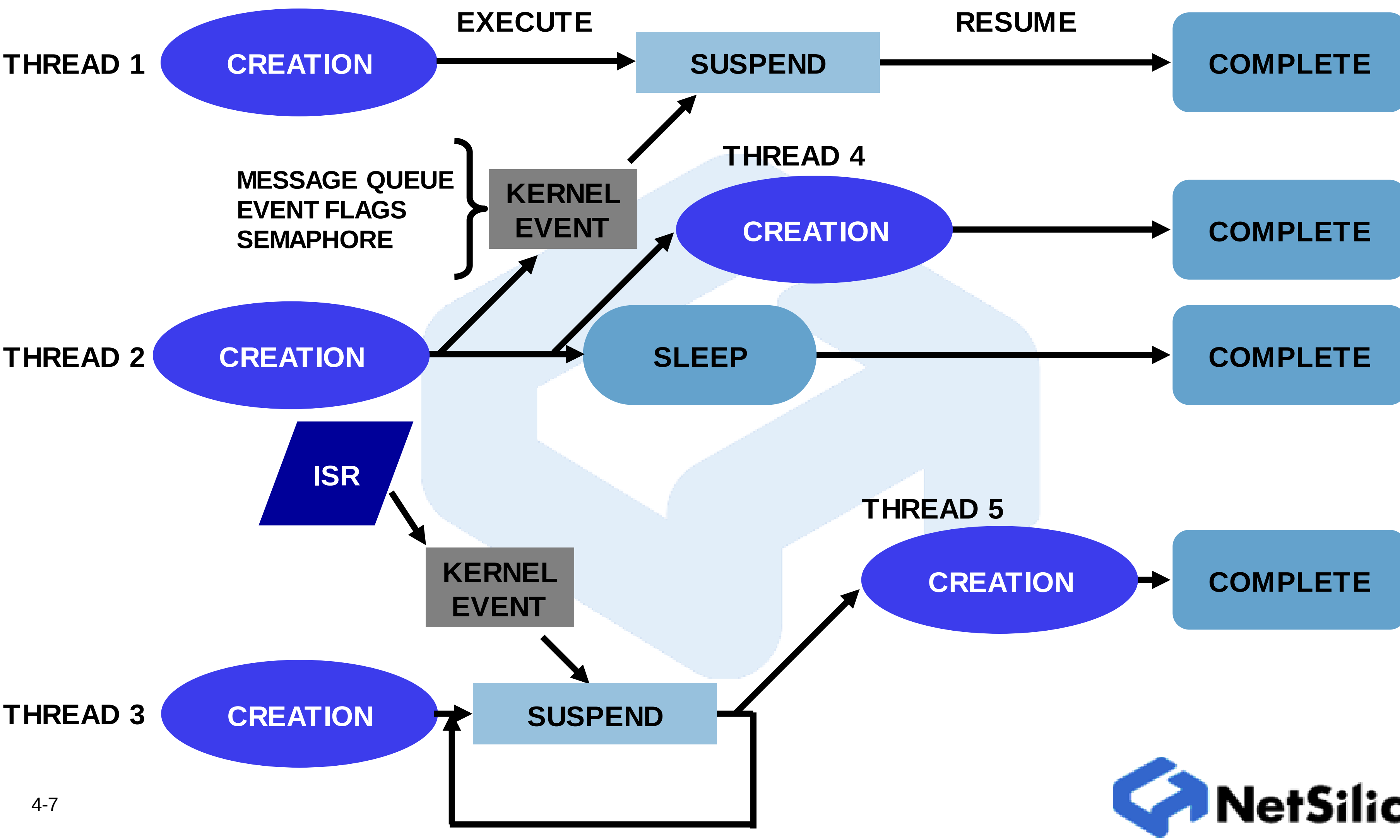
- Thread Creation
  - Easy; One Function Call Spawns Thread
- Thread Management
  - Threads Execute Independently
    - Individual Stack Space
  - Execute Threads Based on Priority
    - 32 Levels of Prioritization
  - Time-Share Equal Priority Threads
    - Round-Robin
- Run-Time Management
  - Locking/Unlocking System Resources
  - Thread-to-Thread Communication
  - Timing
- Interrupt Handling

Thread Introduction

- NET+OS uses threads, not tasks
  - By definition threads share address space, tasks and processes do not
  - In relative terms, overhead associated with thread management is minimal
- NET+OS provides APIs which enable the user to:
  - Create and delete threads
  - Control the execution of threads
  - Control the scheduling of threads

NetSilicon

ThreadX™ Operation



Thread Parameters

- Attributes used to define a thread include:
  - Thread Pointer: Pointer to thread's control block
  - Name Pointer: Pointer to thread's name
  - Entry Function: Pointer to thread's entry-point routine
  - Entry Input: Optional argument passed to thread's entry function.
  - Stack Start: Starting address of thread's stack space in system memory

Thread Parameters (continued)

- Stack Size: Size, in bytes, of the thread’s stack space
- Priority: Relative importance of the thread in relation to other threads
- Preemption Threshold: Highest minimum priority another thread must be before interrupting this thread
- Time Slice: Maximum number of ticks a thread can execute
- Auto Start: Specifies whether the thread should start immediately, or be created in a suspended state

Thread Creation

CRITICAL PARAMETERS

Entry Function

Entry Parameter

Stack Start

Stack Size

Priority

Time Slice

tx_thread_create()

Example Thread Creation: Net+OS Root Thread

From bsproot.c…

#include tx_api.h

```
/*
 * Now create the root thread.  This thread starts NET+OS and the TCP/IP stack.
 */
    ccode = tx_thread_create (&rootThread,                        /* control block for root thread*/
                            "Root Thread",                        /* thread name*/
                            netosStartup,                         /* entry function*/
                            0,                                    /* parameter*/
                            first_unused_memory,                  /* start of stack*/
                            APP_ROOT_STACK_SIZE,                  /* size of stack*/
                            APP_ROOT_PRIORITY,                    /* priority*/
                            APP_ROOT_PRIORITY,                    /* preemption threshold */
                            1,                                    /* time slice threshold*/
                            TX_AUTO_START);                       /* start immediately*/

    if (ccode != TX_SUCCESS)                                      /* if couldn't create thread*/
    {
        netosFatalError ("Unable to create root thread", 1, 1);
    }
}
```

Thread Creation

```
uint tx_thread_create(TX_THREAD *pThread,
                      char *pName,
                      void (*pEntryFunc)(ulong),
                      ulong entryInput,
                      void *pStackStart,
                      ulong stackSize,
                      uint priority,
                      uint preemptThreshold,
                      ulong timeSlice,
                      uint autoStart)
```

- Creates a thread using the specified attributes

NetSilicon

Thread Terminate/Delete

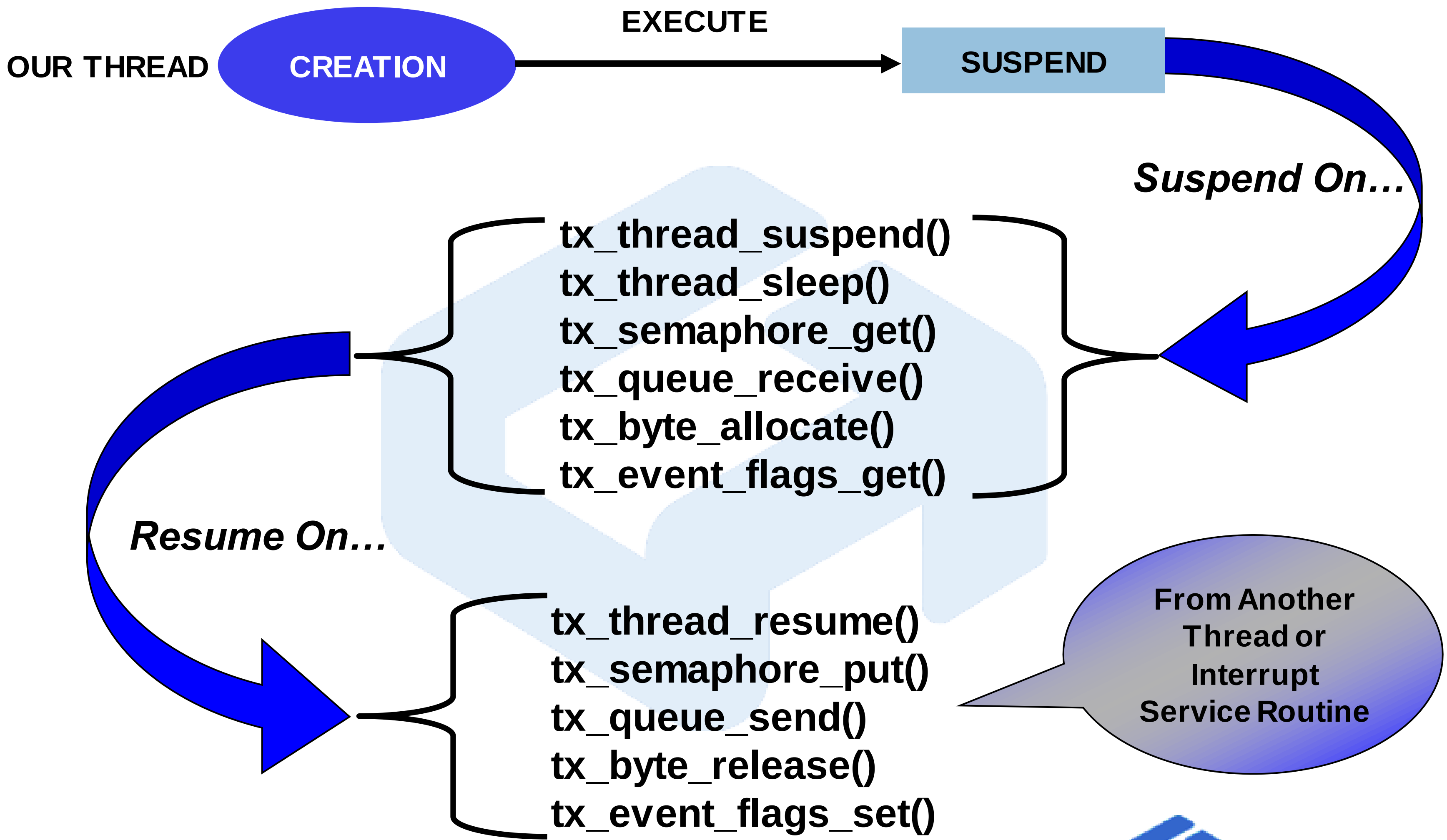
uint **tx_thread_terminate**(TX_THREAD *pThread)

- Terminates the specified thread regardless of state
- Threads are allowed to self terminate
- Terminated threads must be deleted and re-created in order to execute again

uint **tx_thread_delete**(TX_THREAD *pThread)

- Deletes the specified thread, pThread
- Only able to delete threads in the completed (TX_COMPLETED) or terminated (TX_TERMINATED) states
- Application responsible for memory management/cleanup

Thread Suspension / Resumption



NetSilicon

Thread Resume/Suspend

uint **tx_thread_resume**(TX_THREAD *pThread)

- Resumes previously suspended (TX_SUSPENDED) threads
- Resumes threads created without an automatic start

uint **tx_thread_suspend**(TX_THREAD *pThread)

- Suspends the specified thread, pThread
- Threads are allowed to suspend themselves
- Allowed to suspend a currently suspended (TX_SUSPENDED) thread only once

Thread Sleep/Relinquish

uint **tx_thread_sleep**(ulong timerTicks)

- Suspends the calling thread for the specified number of timer ticks, timerTicks

void **tx_thread_relinquish** (void)

- Yields the processor to other threads in the ready (TX_READY) state

NetSilicon

Thread Identify

TX_THREAD* **tx_thread_identify** (void)

- Returns a pointer to the current thread's control block
- When called from an ISR the routine returns the control block of the thread that ran prior to the interrupt

Thread Preemption/Priority/Time Slice Change

uint **tx_thread_preemption_change**(TX_THREAD *pThread, uint newThreshold, uint *pOldThreshold)

- Changes the thread's preemption threshold attribute to the value of newThreshold

uint **tx_thread_priority_change** (TX_THREAD *pThread, uint newPriority, uint *pOldPriority)

- Changes the thread's priority attribute to the value of newPriority
- User will need to adjust the preemption threshold value on their own

uint **tx_thread_time_slice_change**(TX_THREAD *pThread, ulong newTimeSlice, ulong *pOldTimeSlice)

- Changes the thread's time slice attribute to the value of newTimeSlice

Thread Summary

- Thread attributes
  - Maintained in thread control block, TX_THREAD
  - Originally specified in tx_thread_create()
  - Modified with tx_thread_xxx_change() routines
- Thread APIs
  - tx_thread_create() / tx_thread_terminate() / tx_thread_delete()
  - tx_thread_suspend() / tx_thread_resume()
  - tx_thread_sleep() / tx_thread_relinquish()
  - tx_thread_identify()
  - tx_thread_preemption_change()
  - tx_thread_priority_change()
  - tx_thread_time_slice_change()

Thread Summary (continued)

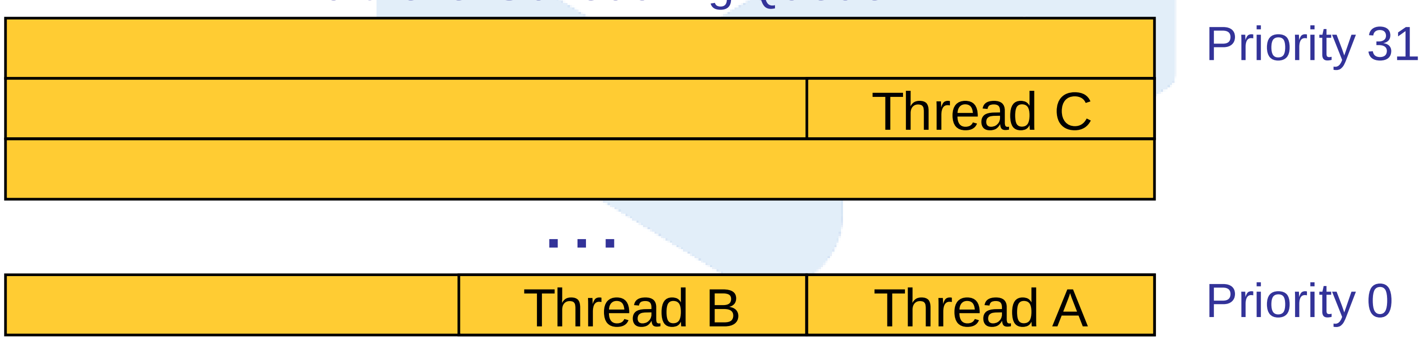
- Thread APIs
  - tx_thread_create() / tx_thread_terminate() / tx_thread_delete()
  - tx_thread_suspend() / tx_thread_resume()
  - tx_thread_sleep() / tx_thread_relinquish()
  - tx_thread_identify()
  - tx_thread_preemption_change()
  - tx_thread_priority_change()
  - tx_thread_time_slice_change()

ThreadX Scheduler

Multilevel Queue Scheduler

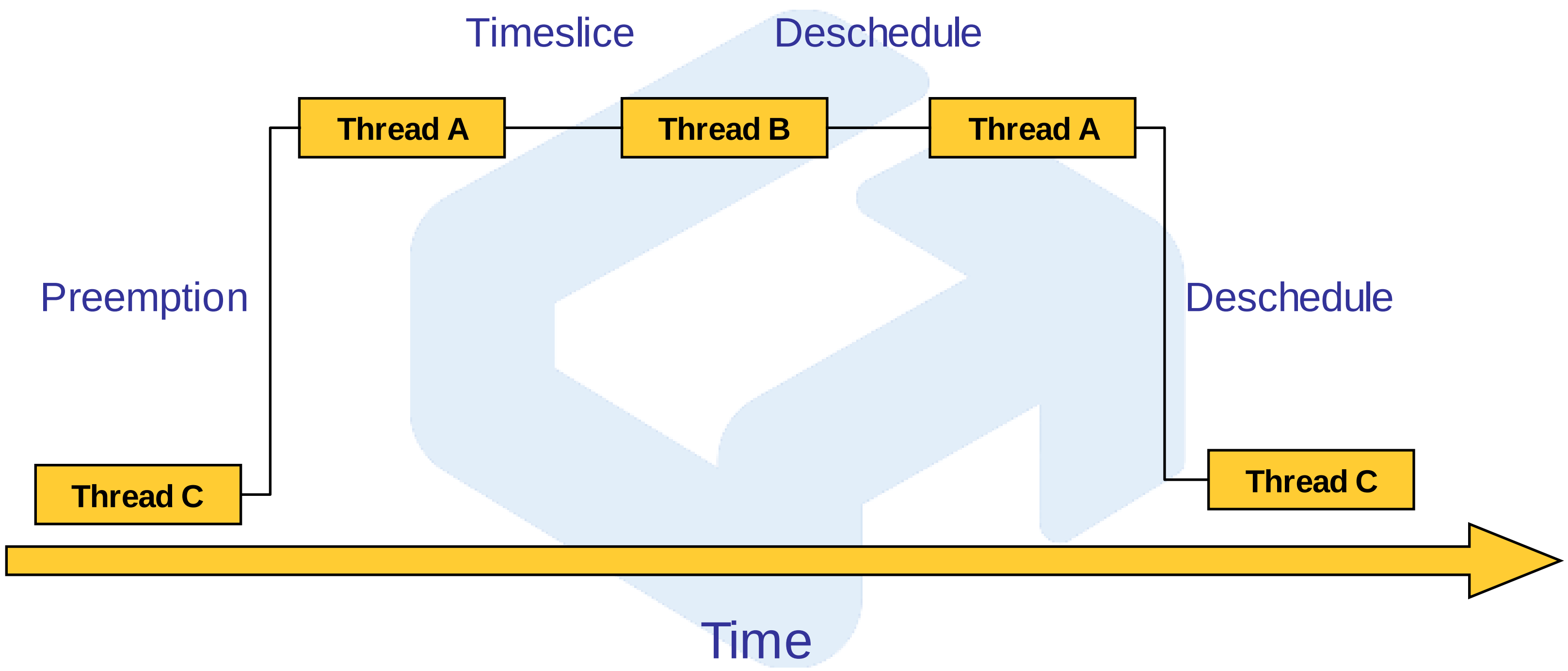
Scenario:
Thread A, Thread B, and Thread C are configured during creation w/ priority levels 0, 0, 30, respectively.

Multilevel Scheduling Queue

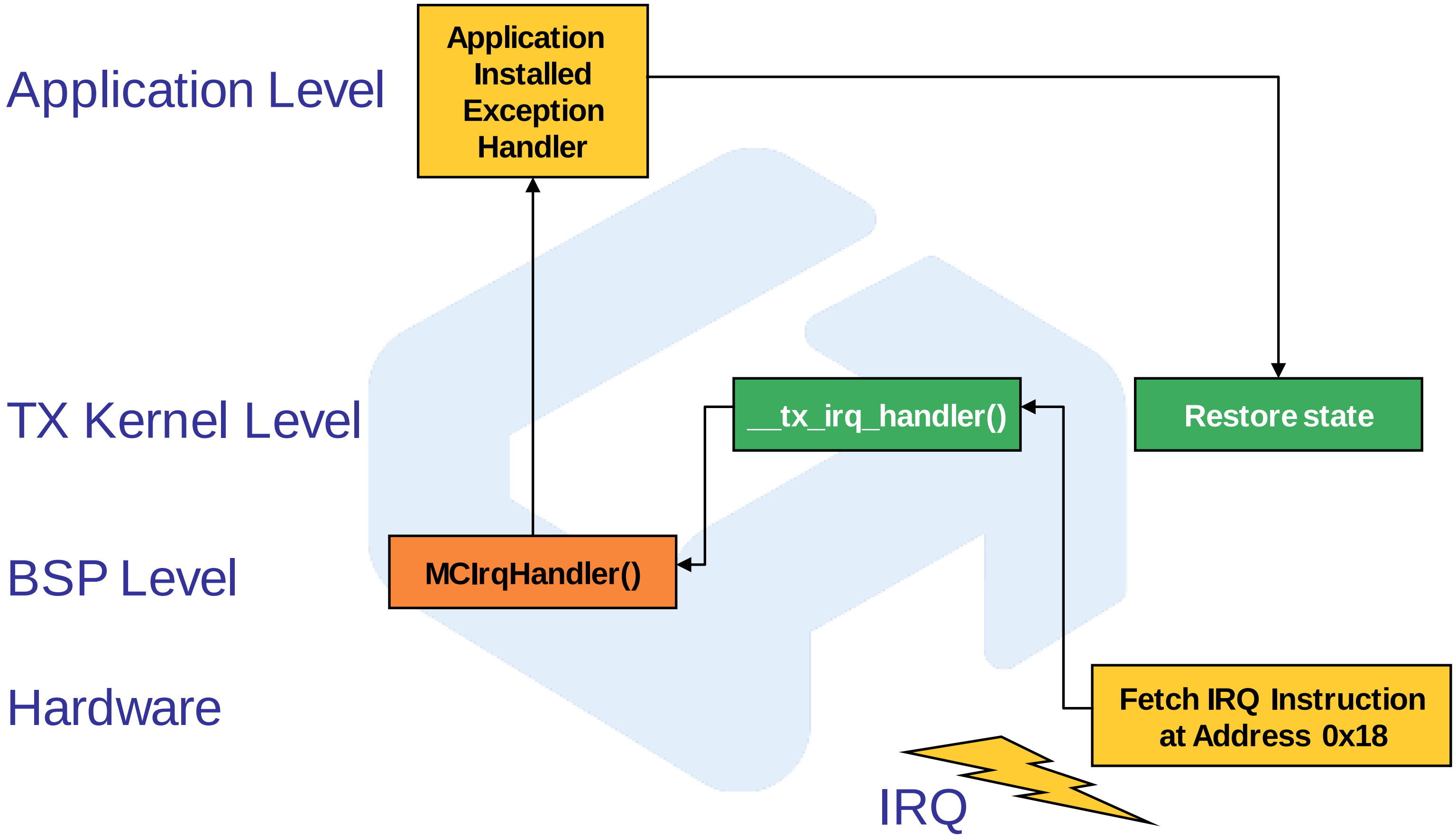


ThreadX Scheduler (cont.)

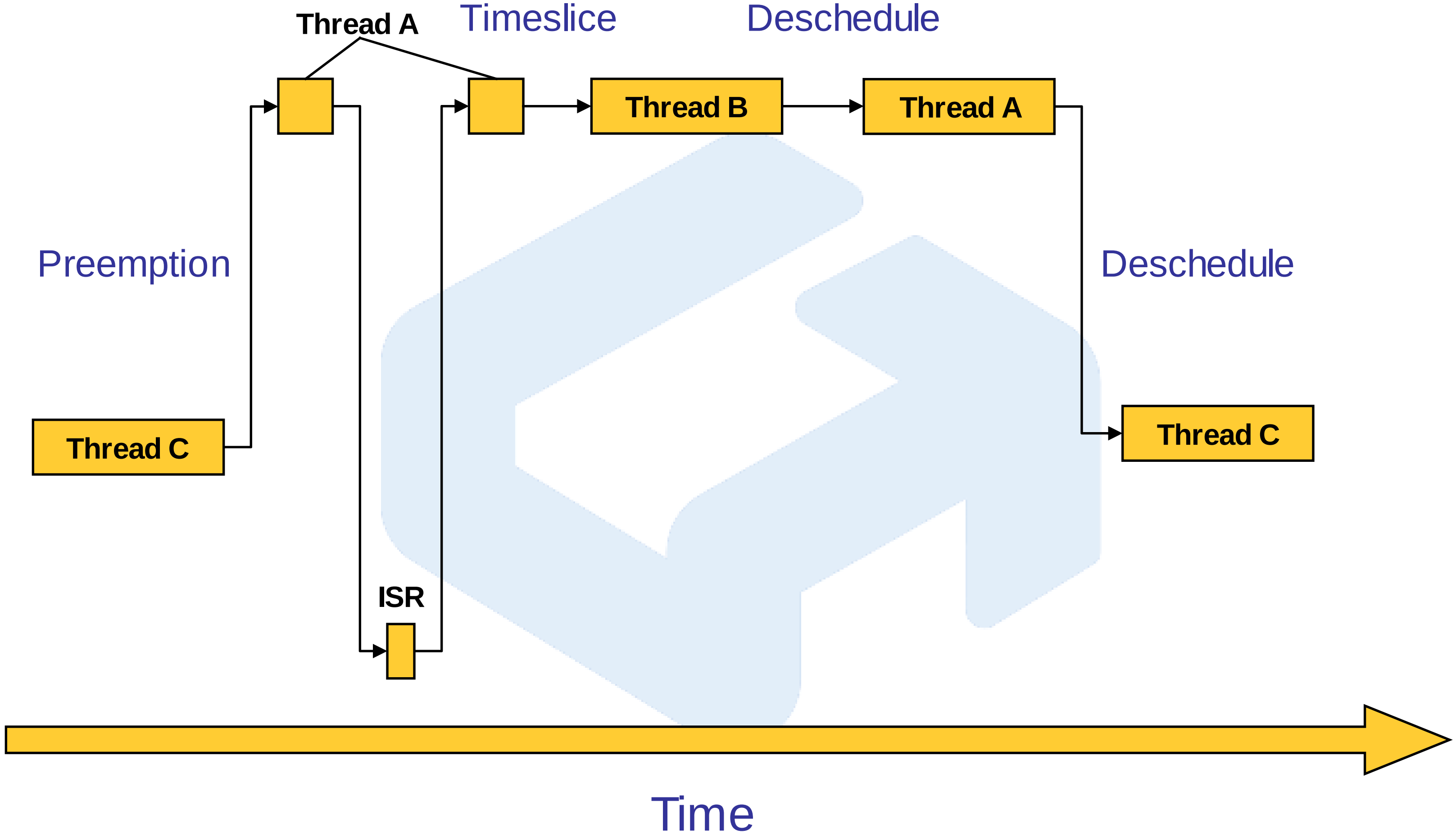
Scheduler Thread Operation:



ThreadX Exception Handling



ThreadX Exception Handling cont.



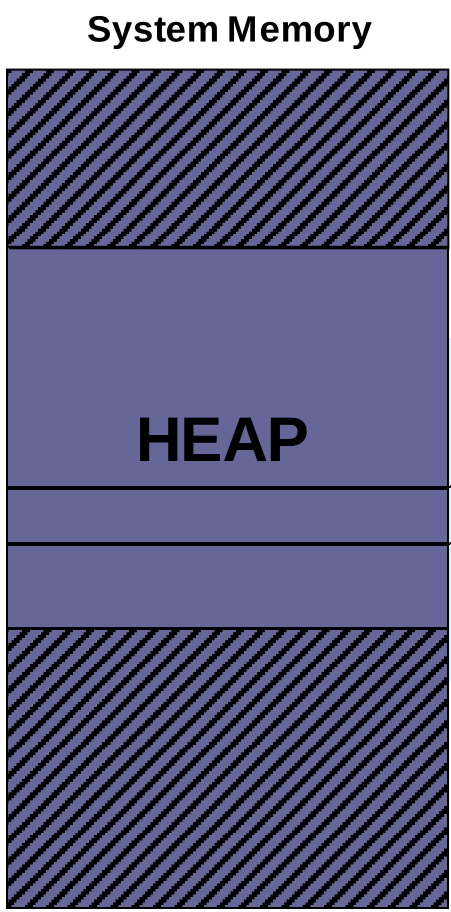
NetSilicon

4.2 Memory Management

Memory Management - Block Pools

- Block Pools
  - Deterministic allocation/free time
  - Not subject to memory fragmentation
  - Pools must be sized to handle worst case memory scenario
  - Publicly available resource
  - Overhead associated with each block equal to a C-pointer
  - Pool control blocks, TX_BLOCK_POOL, often defined globally
  - Pools can be located anywhere in memory

Memory Block Pools - Creation



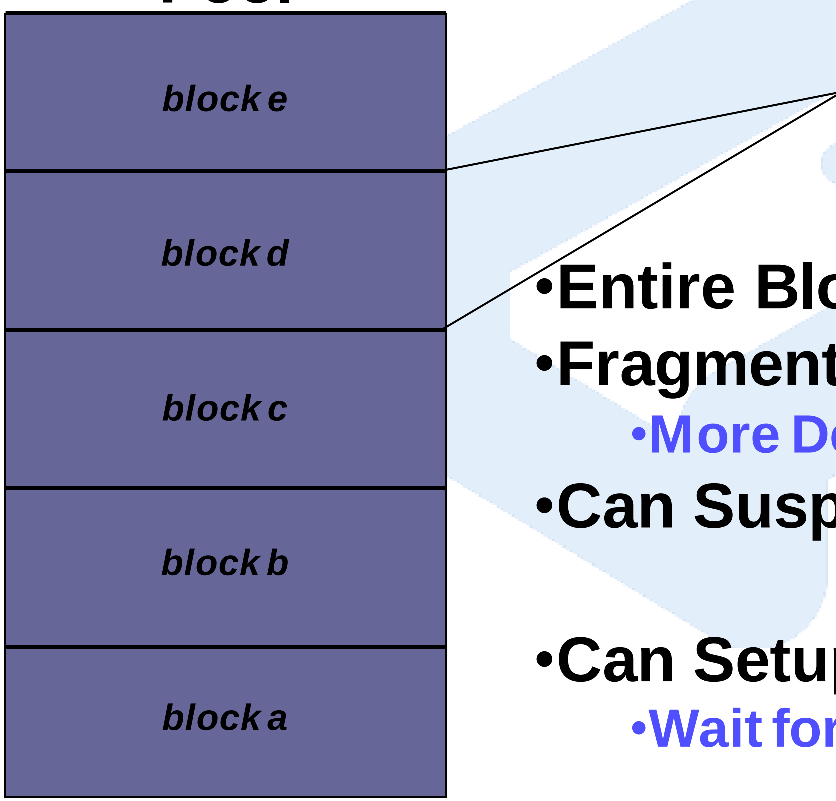
tx_block_pool_create()

- **Total Size in Bytes**
- **Divided Into Equal-size Sub-blocks**
- **Can Suspend if Unsuccessful**
- **Flexibility in Location**
    - **Programmer Chooses Start Address**

NetSilicon

Memory Block Pools - Allocation

Memory Block Pool



tx_block_allocate()

- **Entire Block Allocated**
- **Fragmentation Less Likely**
  - **More Deterministic than malloc()**
- **Can Suspend if Unsuccessful**

-OR-

- **Can Setup Time-Out Period**
  - **Wait for Another Thread to Free a Block**

NetSilicon

Block Pool Create/Delete

uint **tx_block_pool_create**(TX_BLOCK_POOL *pPool, char *pName, ulong blockSize, void *pPoolStart, ulong poolSize)

- Creates a pool of fixed size memory blocks, pPool
- total blocks = (poolSize)/ (blockSize + sizeof(void *))

uint **tx_block_pool_delete**(TX_BLOCK_POOL *pPool)

- Deletes a pool of fixed size memory blocks, pPool
- Threads (suspended) waiting for memory from this pool are resumed and given a TX_DELETED status
- Application's responsibility to prevent threads from using memory in the former block pool's memory region

NetSilicon

Block Pool Allocate/Release

uint **tx_block_allocate**(TX_BLOCK_POOL *pPool, void **pBlock, ulong waitOption)

- Allocates a fixed size memory block from the pool pointed to by pPool
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000)
  - TX_WAIT_FOREVER (0xFFFF FFFF)
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE)

uint **tx_block_release**(void *pBlock)

- Releases the previously allocated block, pBlock, into the memory pool
- Application's responsibility to prevent threads from using released memory

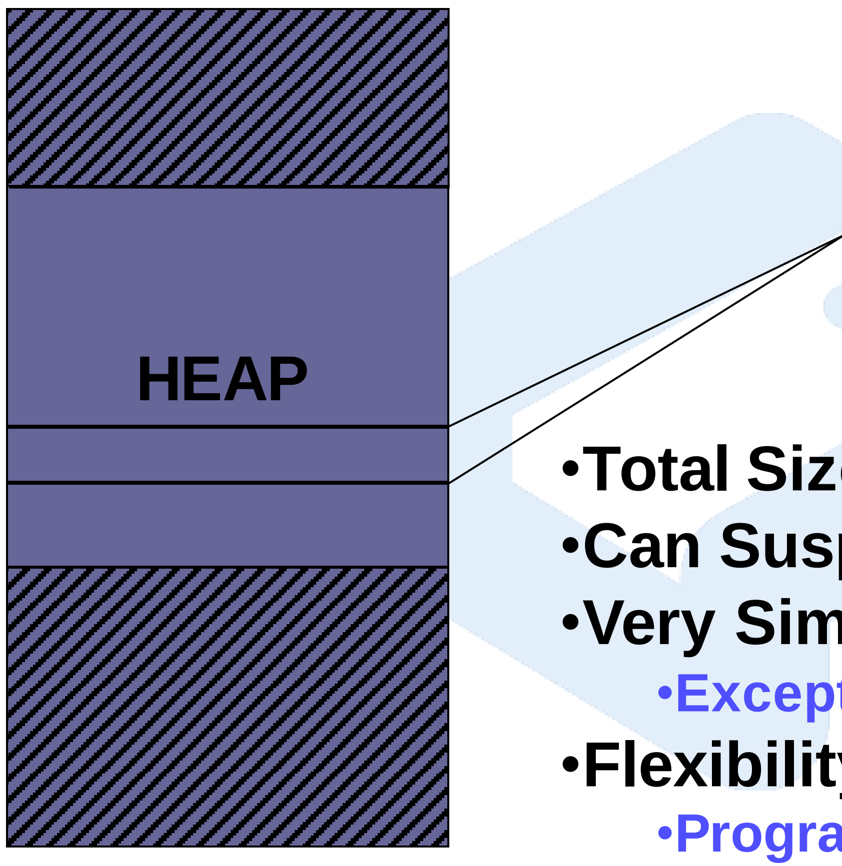
Memory Management - Byte Pools

- Byte Pools
  - Non-deterministic
  - Memory allocated using ‘First-fit’ algorithm
  - Suffers from memory fragmentation
  - Publicly available resource but should *NOT* be used in ISRs
  - Pool control blocks, TX_BYTE_POOL, often defined globally

NetSilicon

Memory Byte Pools - Creation

System Memory

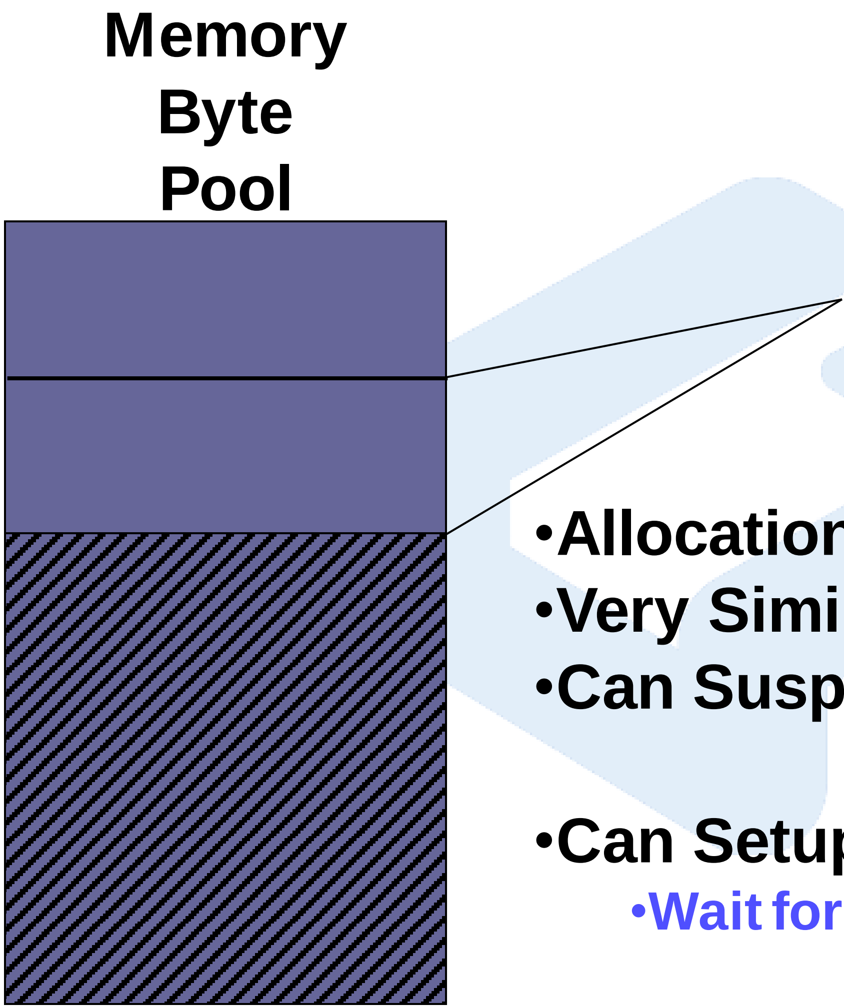


tx_byte_pool_create()

- **Total Size in Bytes**
- **Can Suspend if Unsuccessful**
- **Very Similar to Block Pools**
  - **Except no Fixed-Sized Blocks**
- **Flexibility in Location**
  - **Programmer Chooses Start Address**

NetSilicon

Memory Byte Pools - Allocation



- Allocation Specified in Bytes
- Very Similar Concept as malloc()
- Can Suspend if Unsuccessful

  -OR-

- Can Setup Time-Out Period
  - Wait for Another Thread to Free a Block

Byte Pool Create/Delete

uint **tx_byte_pool_create**(TX_BYTE_POOL *pPool, char *pName, void *pPoolStart, ulong poolSize)

- Creates a memory pool in the specified area, pPoolStart
- Initially consists of one memory block, of size poolSize
- Broken into smaller blocks during de-fragmentation process

uint **tx_byte_pool_delete**(TX_BYTE_POOL *pPool)

- Deletes the byte pool, pPool
- Threads (suspended) waiting for memory from this pool are resumed and given a TX_DELETED status
- Application's responsibility to prevent threads from using memory in the former byte pool's memory region

Byte Pool Allocate/Release

uint **tx_byte_allocate**(TX_BYTE_POOL *pPool, void **pMemory, ulong memorySize, ulong waitOption)

- Allocates memorySize bytes from the pool pointed to by pPool
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000)
  - TX_WAIT_FOREVER (0xFFFF FFFF)
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE)
- Performance a function of pool fragmentation, non-deterministic

uint **tx_byte_release**(void *pMemory)

- Releases the previously allocated memory, pMemory, back into the pool
- Application's responsibility to prevent threads from using released memory

Memory Management Summary

- Memory Management attributes
  - Two different types of memory allocation, fixed-size blocks and heap
  - Memory allocation information maintained in memory control blocks, TX_BLOCK_POOL or TX_BYTE_POOL

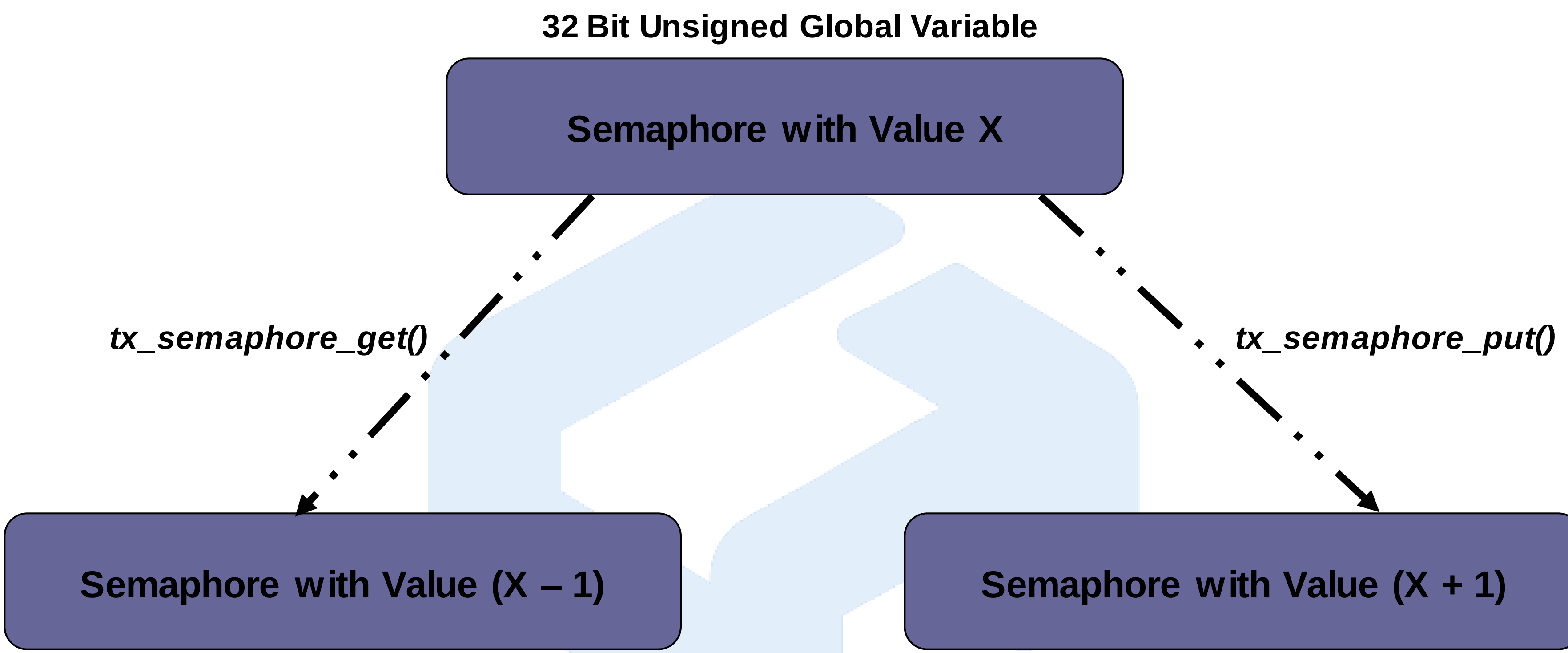
- Memory Management APIs
  - tx_block_pool_create() / tx_block_pool_delete()
  - tx_block_allocate() / tx_block_release()
  - tx_byte_pool_create() / tx_byte_pool_delete()
  - tx_byte_allocate() / tx_byte_release()

4.3 Semaphores

Semaphores

- ThreadX supports 32-bit counting semaphores (4,294,967,296!)
- Typically used for mutual exclusion, can also be applied to event notification
- Initializing a semaphore’s initial count to 1 creates a ‘binary semaphore’
- Semaphores must be used carefully in order to avoid deadlocks or priority inversion
- Semaphore control blocks, TX_SEMAPHORE, often defined globally
- Semaphores can be located anywhere in memory

Counting Semaphores



NetSilicon

Semaphore Create/Delete

uint **tx_semaphore_create**(TX_SEMAPHORE *pSemaphore, char *pName, ulong initialCount)

- Creates a counting semaphore
- Semaphore's count initialized to initialCount

uint **tx_semaphore_delete**(TX_SEMAPHORE *pSemaphore)

- Deletes the specified semaphore, pSemaphore
- Application's responsibility to prevent threads from using a deleted semaphore
- Threads (suspended) waiting for this semaphore are resumed and given a TX_DELETED status

NetSilicon

Semaphore Get/Put

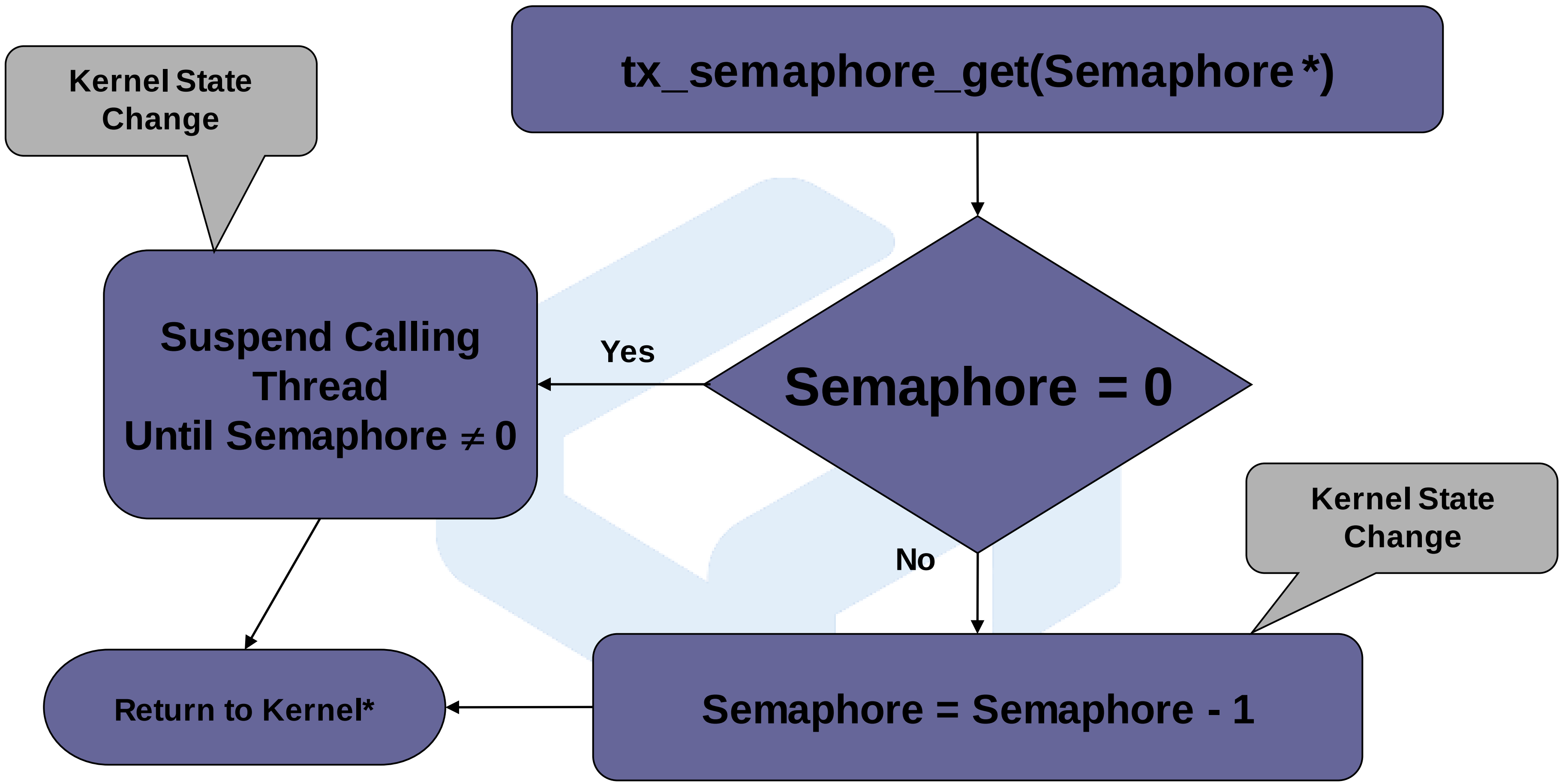
uint **tx_semaphore_get**(TX_SEMAPHORE *pSemaphore, ulong waitOption)

- Retrieves an instance of the specified semaphore, pSemaphore, depending upon availability and waitOption
- Semaphore’s count decremented by 1
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000) – Return Immediately
  - TX_WAIT_FOREVER (0xFFFF FFFF) – Block, waiting for the semaphore
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE) – Block, waiting for the specified time

uint **tx_semaphore_put** (TX_SEMAPHORE *pSemaphore)

- Puts an instance of the specified semaphore, pSemaphore
- Semaphore’s count incremented by 1
- If a semaphore’s count is 0xFFFF FFFF, new count will be 0

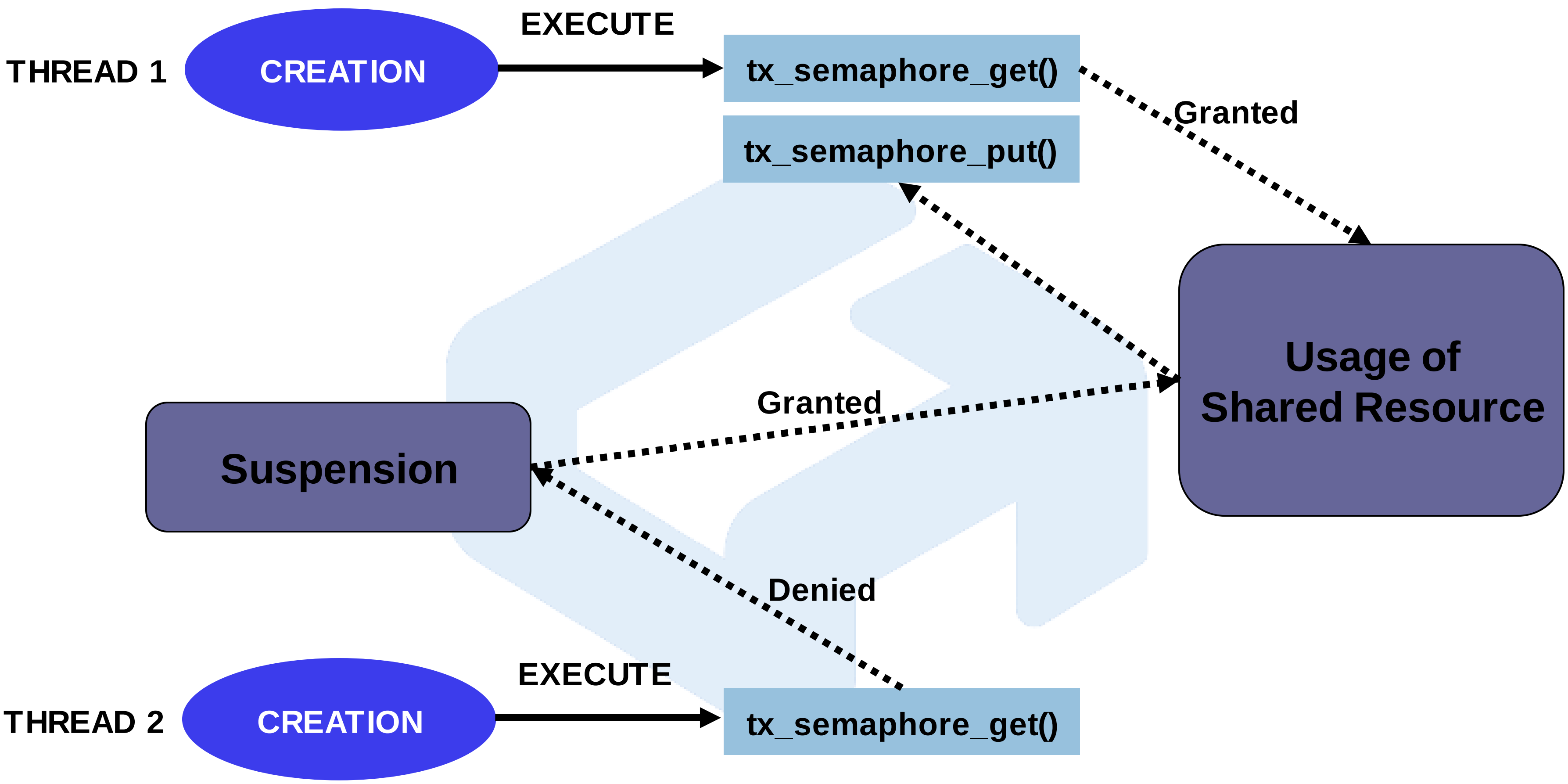
A Closer Look At tx_semaphore_get()



*Kernel Decides which Thread to Resume

NetSilicon

Semaphores in Action



Example Semaphore Usage

Declaration `TX_SEMAPHORE serial;`

Initial Value
'1' for Binary

Creation `status = tx_semaphore_create (&serial, "Serial Port Use", 1);`

Name

Get `tx_semaphore_get (&serial, TX_WAIT_FOREVER);`

Wait
Condition

Put `tx_semaphore_put (&serial);`

Semaphore Summary

- Semaphore attributes
  - Semaphore information maintained in semaphore control blocks, TX_SEMAPHORE
  - No built in support for deadlock avoidance or priority-inversion handling
- Semaphore APIs
  - tx_semaphore_create() / tx_semaphore_delete()
  - tx_semaphore_get() / tx_semaphore_put()

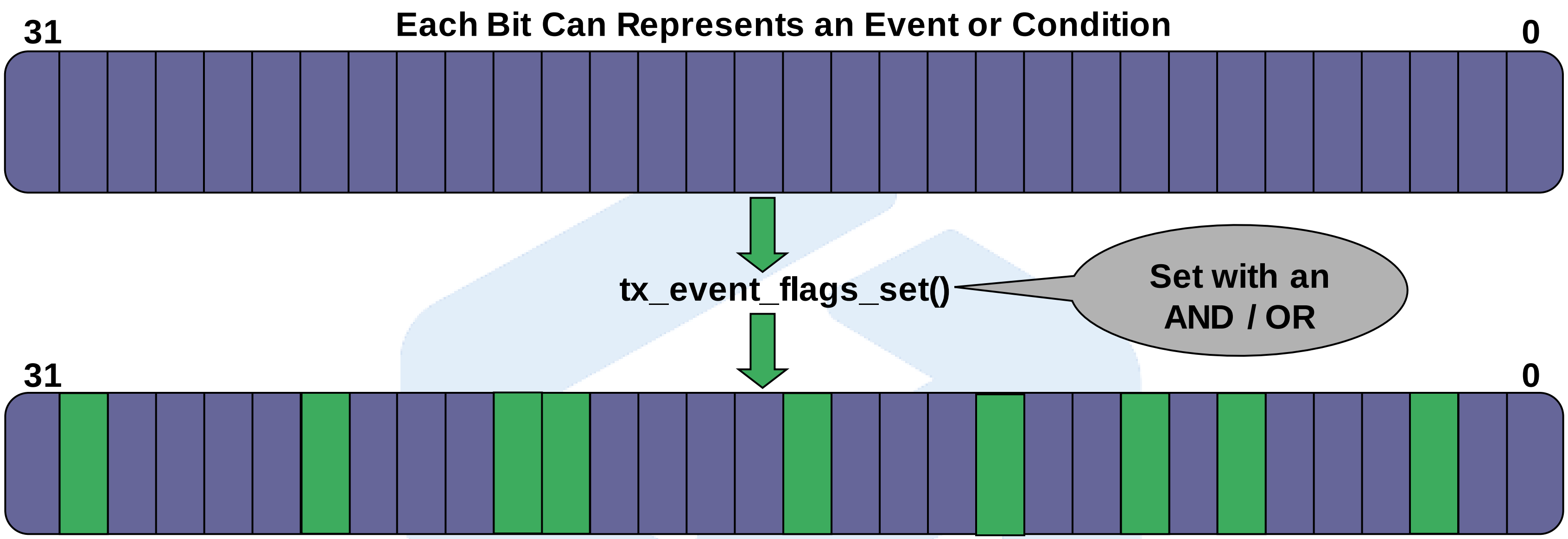
4.4 Event Flags

Event Flags

- Event flags provide a means of thread synchronization
- Classified in groups of 32, making up a single word
- When a group is created, all 32 flags are initialized to 0
- Multiple threads can use the same group
- Event flag control blocks, TX_EVENT_FLAGS_GROUP, often defined globally
- Event flag groups can be located anywhere in memory

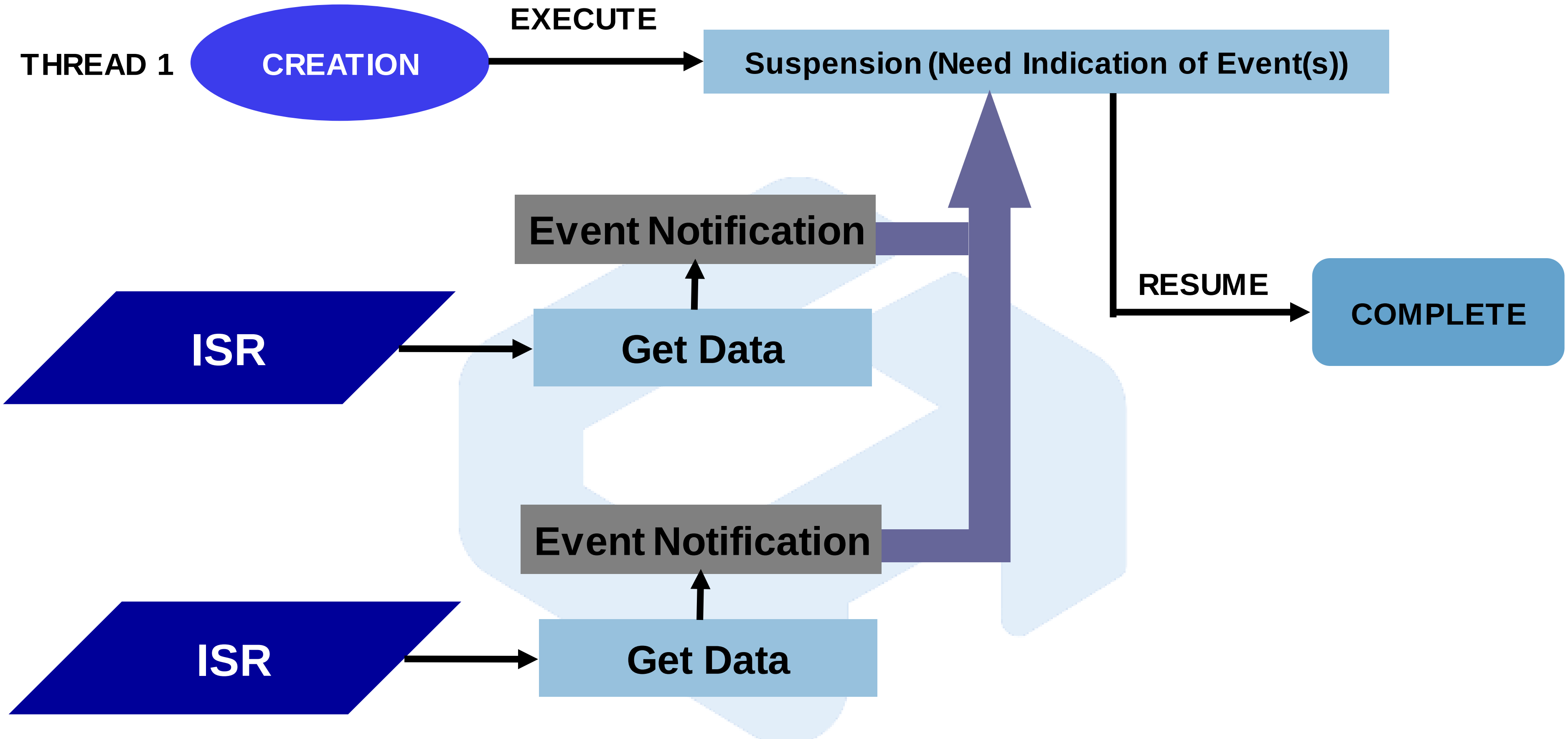
Event Flags

32 Bit Unsigned Global Variable



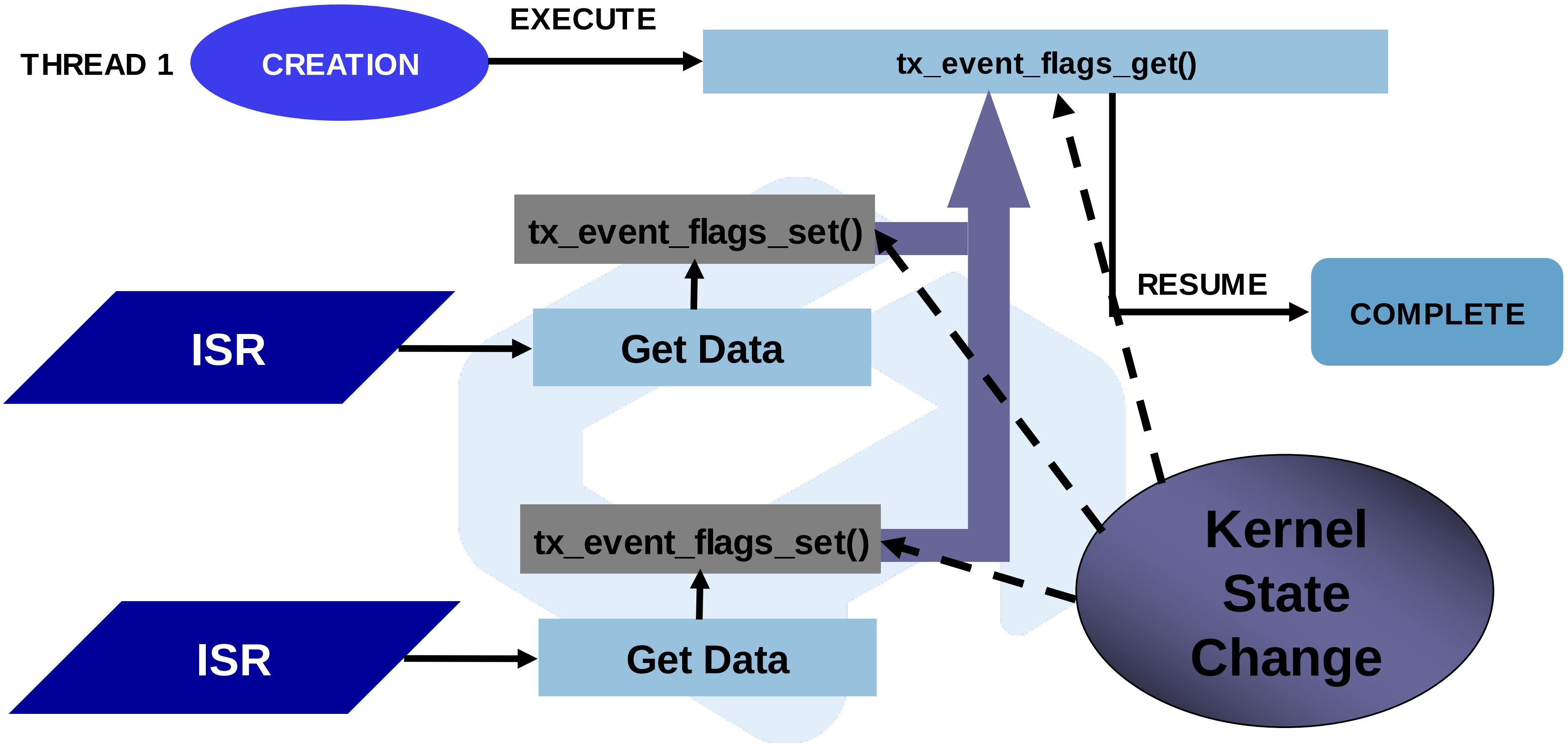
The State of Event Flags Retrieved with tx_event_flags_get()

Event Flags - Conceptual



NetSilicon

Event Flags-Implementation



Example Event Flags

Declaration TX_EVENT_FLAGS_GROUP tcp_sleep_events;

Name

Creation tx_event_flags_create (&tcp_sleep_events, "TCP Sleep Event Flags");

Set tx_event_flags_set(&tcp_sleep_events,(1 << i),TX_OR);

Flag Mask

Modification Option

Get tx_event_flags_get(&tcp_sleep_events,(1 << i),TX_OR_CLEAR, &flags, TX_WAIT_FOREVER);

Flags Copied Here

Wait Option

Event Flags Create/Delete

uint **tx_event_flags_create**(TX_EVENT_FLAGS_GROUP *pGroup, char *pName)

- Creates a group of 32 flags, all initialized to 0

uint **tx_event_flags_delete**(TX_EVENT_FLAGS_GROUP *pGroup)

- Deletes the specified event flag group, pGroup
- Application's responsibility to prevent use of a deleted event flag group
- Threads (suspended) waiting for events from this group are resumed and given a TX_DELETED status

Event Flags Get

uint **tx_event_flags_get** (TX_EVENT_FLAGS_GROUP *pGroup, ulong requestedFlags, uint getOption, ulong *pActualFlags, ulong waitOption)

- Retrieves event flags from the specified group, pGroup
- The getOption parameter allows control over which flags are returned, and possibly cleared
- Possible values for getOption include:
  - TX_AND
  - TX_AND_CLEAR
  - TX_OR
  - TX_OR_CLEAR
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000) – Return Immediately
  - TX_WAIT_FOREVER (0xFFFF FFFF) – Block, waiting for the event
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE) – Block, waiting for the specified time

Event Flags Set

uint **tx_event_flags_set** (TX_EVENT_FLAGS_GROUP *pGroup, ulong flagsToSet, uint setOption)

- Sets/clears event flags in the specified group, pGroup, depending upon the setOption
- Performs the setOption between the current flags and flagsToSet
- Possible values for setOption include:
  - TX_AND
  - TX_OR

Event Flags Summary

- Event Flag attributes
  - Event Flag information maintained in event flag control blocks, TX_EVENT_FLAGS_GROUP
  - Multiple threads and use (and be suspended on) the same event flag group
- Event Flags APIs
  - tx_event_flags_create() / tx_event_flags_delete()
  - tx_event_flags_get() / tx_event_flags_set ()

4.5 Message Queues

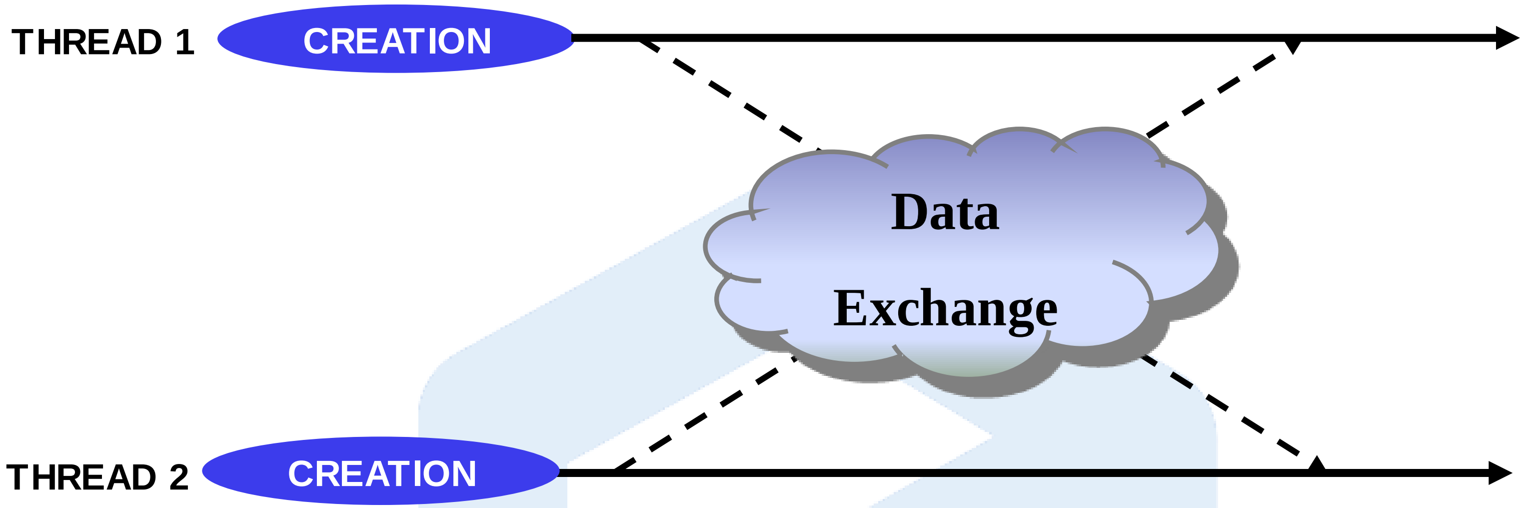
NetSilicon

Message Queues

- Primary means of inter-thread communication in ThreadX
- Message queues that hold a single message referred to as a ‘mailbox’
- Queues support messages of 1, 2, 4, 8, or 16 32-bit sizes
  - Anything larger should use pointers
- Messages are copied to and from queues
  - In the case of a suspended thread, message sent directly to queue
- Queue control blocks, TX_QUEUE, often defined globally
- Queues can be located anywhere in memory
- To prevent memory corruption the application *must* ensure that its receiving memory is at least as large as the message size

NetSilicon

Message Queues

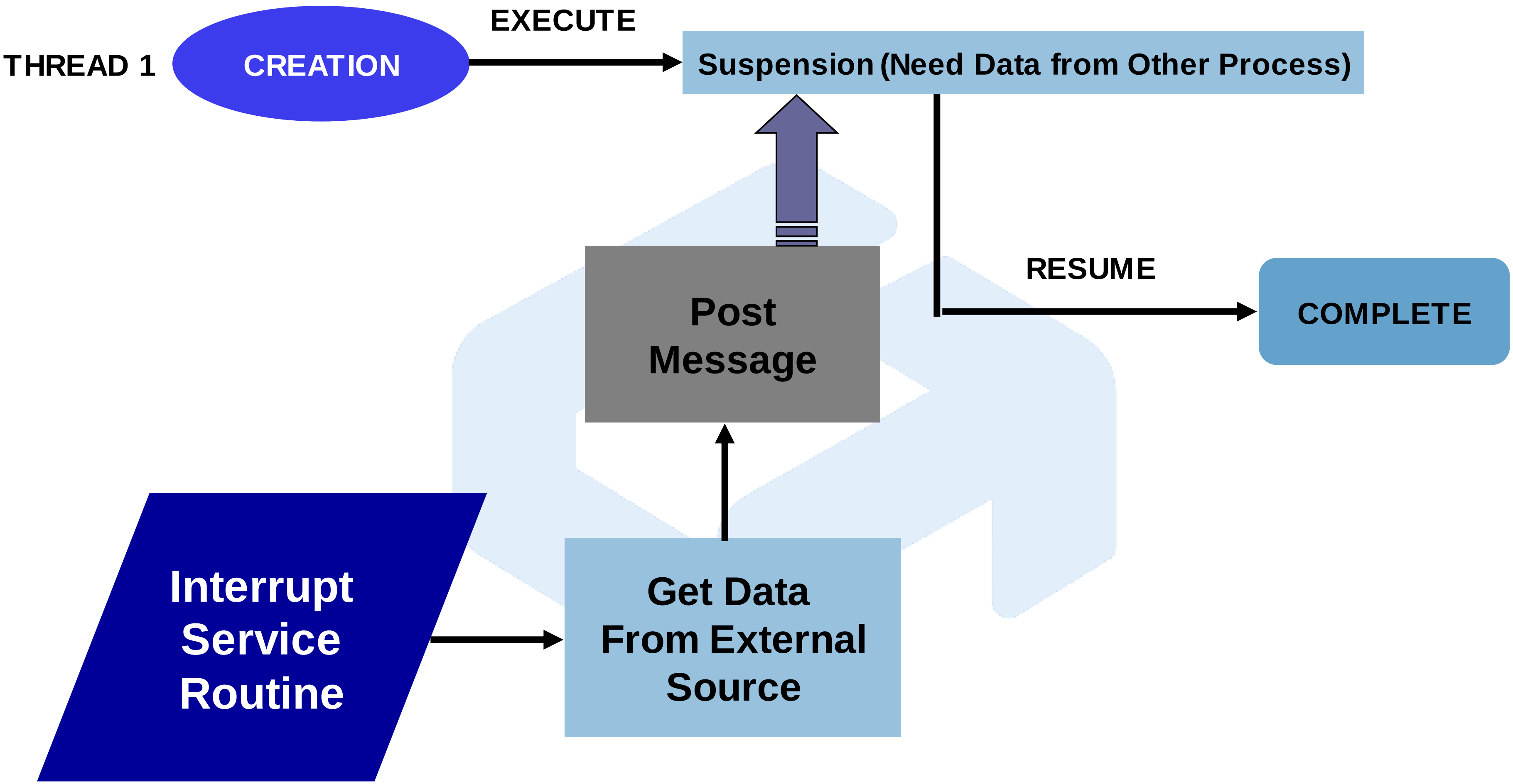


Message Queues Provide a Convenient Means of InterThread Communication

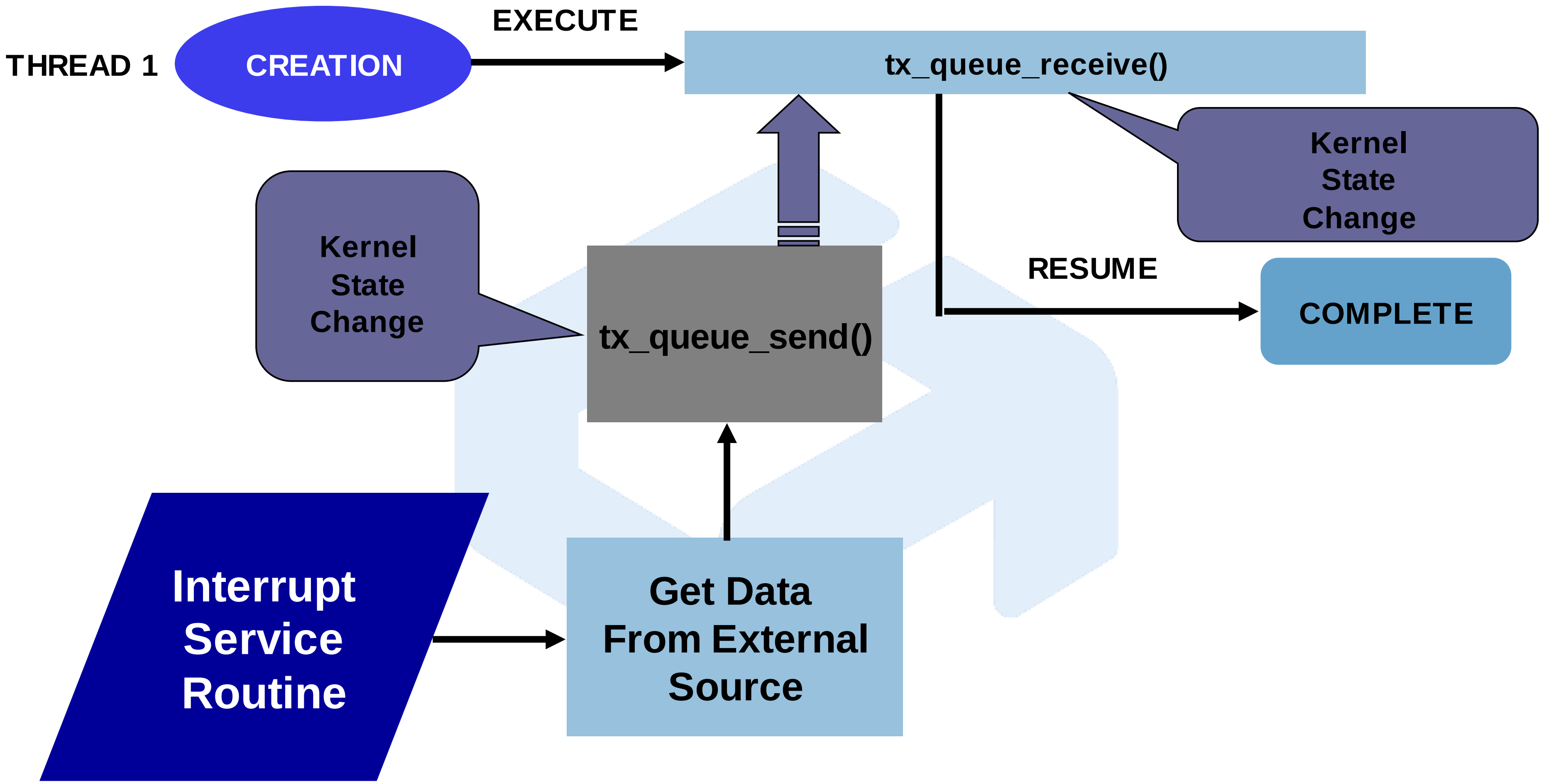
Data Exchange is Done by Posting and Receiving Data from Shared 'Mailboxes'

NetSilicon

Message Queues - Conceptual



Message Queues-Implementation



NetSilicon

Example Message Queue

Declaration **TX_QUEUE Data_Queue;**

Name

Creation **status = tx_queue_create (&Data_Queue, "UDP Send Data", TX_2_ULONG, Message_Pointer, 8);**

Mailbox Location

Message Size

Send **status = tx_queue_send (&Data_Queue, package, TX_WAIT_FOREVER);**

Data

Wait Condition

Receive **tx_queue_receive(&Data_Queue, package, TX_WAIT_FOREVER);**

Message Queue Send/Receive

uint **tx_queue_send**(TX_QUEUE *pQueue, void *pSource, ulong waitOption)

- Sends a message to pQueue
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000)
  - TX_WAIT_FOREVER (0xFFFF FFFF)
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE)

uint **tx_queue_receive**(TX_QUEUE *pQueue, void *pDestination, ulong waitOption)

- Retrieves a message from pQueue
- Possible values for waitOption include:
  - TX_NO_WAIT (0x0000 0000)
  - TX_WAIT_FOREVER (0xFFFF FFFF)
  - time-out value, in ticks (0x0000 0001 - 0xFFFF FFFE)

Message Queue Flush

uint **tx_queue_flush**(TX_QUEUE *pQueue)

- Deletes all messages in the queue, pQueue
- Threads (suspended) waiting for message from this queue are resumed, and given a status indicating the message send was successful

Queue Summary

- Message Queue attributes
  - Queue information maintained in queue control blocks, TX_QUEUE
  - Size specified during call to tx_queue_create()
  - Queues implemented as FIFOs
- Message Queue APIs
  - tx_queue_create() / tx_queue_delete()
  - tx_queue_send() / tx_queue_receive()
  - tx_queue_flush()

4.6 Timers

Timers

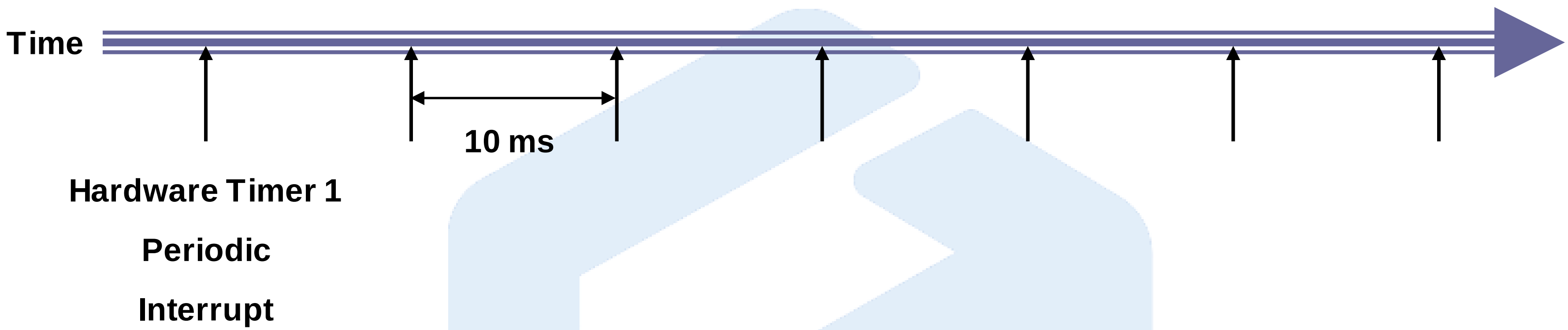
- ThreadX supports both one-shot and periodic timers
- Hardware must generate periodic interrupts for proper timer functionality
- Timers are executed in the order they become active
- Timer control blocks, TX_TIMER, often defined globally
- Timers can be located anywhere in memory
- A Hardware Timer is used to derive the System Tick Rate
- The Timer is fix and can not be changed
- The System Tick Rate determines how oftenthe Scheduler will be called
- The System Tick Rate is set with #define
  BSP_TICKS_PER_SECOND which defaults to 100

Note: Increasing the Tick Rate increases the CPU time spent inside the Scheduler

NetSilicon

Timer Functionality

Background Information – Net+OS Operation

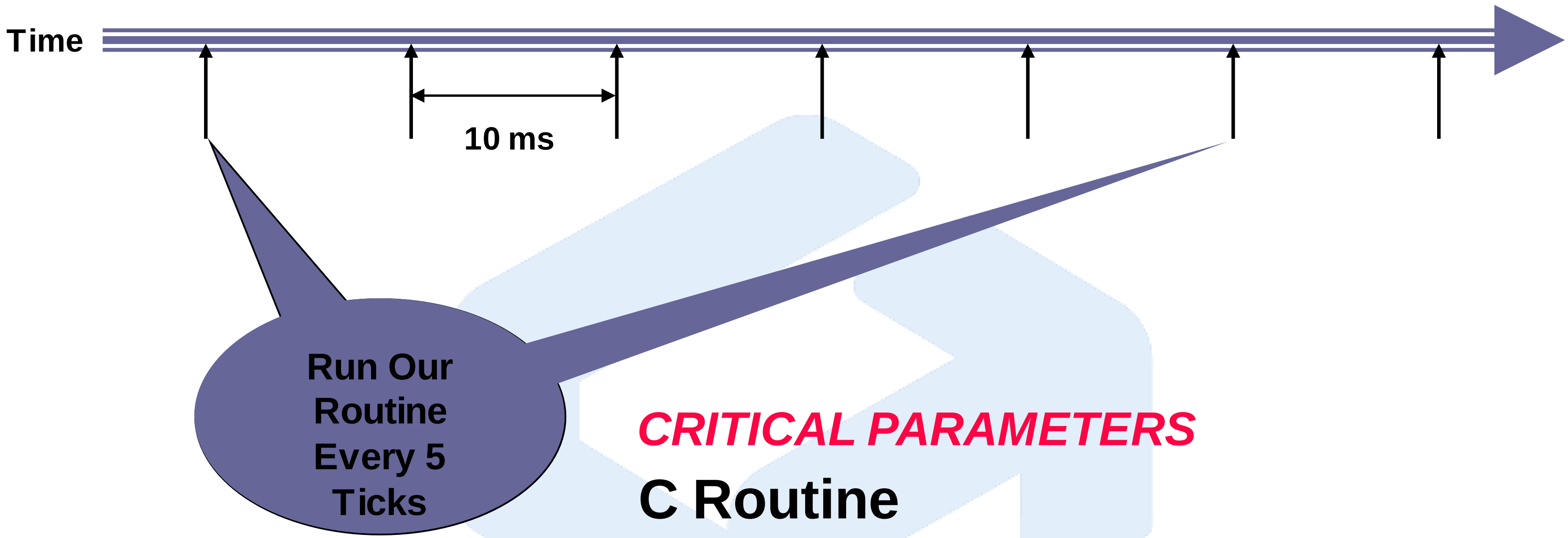


This is How the Operating System Handles All Time-Related Functionality.

- **Timeouts**
- **Sleeps**
- **Time Slices**

NetSilicon

Application Timers



CRITICAL PARAMETERS

C Routine
Entry Parameter
Start Delay
Frequency

tx_timer_create()

NetSilicon

Timer Create/Delete

uint **tx_timer_create**(TX_TIMER *pTimer, char *pName, void *pExpirationFunc(ulong), ulong expirationInput, ulong initialTicks, ulong rescheduleTicks, uint autoActivate)

- Creates a timer that executes pExpirationFunc upon timer expiry
- Initial ticks can range from 0x0000 0001 – 0xFFFF FFFF
- Timer created in TX_AUTO_ACTIVATE or TX_NO_ACTIVATE states
- Assign 0 to rescheduleTicks to make the timer 'one-shot', otherwise rescheduleTicks indicates the timer period *after* the first period

uint **tx_timer_delete**(TX_TIMER *pTimer)

- Deletes the specified timer, pTimer
- Application's responsibility to prevent threads from using a deleted timer

Timer Activate/Deactivate/Change

uint **tx_timer_activate** (TX_TIMER *pTimer)

– Activates the specified timer, pTimer

uint **tx_timer_deactivate** (TX_TIMER *pTimer)

– Deactivates the specified timer, pTimer

uint **tx_timer_change**(TX_TIMER *pTimer, ulong initialTicks, ulong rescheduleTicks)

– Changes the expiration attributes of the specified timer, pTimer
– Timer must be deactivated before calling this routine, and activated later to restart

Timer Summary

- Timer attributes
  - Timer information maintained in timer control blocks, TX_TIMER
  - Timers can have a different initial expiry time than their periodic rate
- Timer APIs
  - tx_timer_create() / tx_timer_delete()
  - tx_timer_activate() / tx_time_deactivate()
  - tx_timer_change()

System Tick Timer Implementation

- Initialization takes place in bsptimer.c in functionnetosSetupSystemClock()

```
void netosSetupSystemClock (void)
{
   /* Declaration for standard ThreadX timer ISR. */
   extern int _tx_timer_interrupt (void *);
#ifdef NS9750
   MCSetTimerClockSelect(THREADX_TIMER, 0);    /* set timer clock #0 select to CPU */
   MCSetTimerMode(THREADX_TIMER,0);            /* set timer clock #0 to internal timer */
   MCSetTimerInterruptSelect(THREADX_TIMER,1); /* enable timer clock #0 interrupt */
   MCSetTimerUpDownSelect(THREADX_TIMER,1);    /* set timer clock #0 as a downcounter */
   MCSetTimerBit(THREADX_TIMER, 1);            /* set timer as 32 bit timer */
   MCSetTimerReloadEnable(THREADX_TIMER, 1);   /* set timer reload enable */
   MCReloadTimerCounter(THREADX_TIMER, THREADX_TIMER0_RELOAD_VALUE);

   /* netosInstallTimer0Isr (_tx_timer_interrupt); */
   MCInstallIsr(TIMER0_INTERRUPT, _tx_timer_interrupt, NULL);
   MCEnableTimer(THREADX_TIMER);
#else
...
```

4.7 Other Kernel Services

Other Kernel Services

- Allows the user to initialize the system clock to a known value
- Allows the user to get the current value of the system clock
- Allows the user to enable/disable interrupts

Time Get/Set

ulong **tx_time_get** (void)

- Returns the contents of the internal system clock
- System clock initialized to 0 during system startup

void **tx_time_set**(ulong newTimer)

- Sets the internal system clock
- Valid ranges are 0x0000 0000 – 0xFFFF FFFF

NetSilicon

Interrupt Control

uint **tx_interrupt_control**(uint interruptPosture)

- Enables or disables interrupts depending upon the input, TX_INT_ENABLE or TX_INT_DISABLE
- The interruptPosture is part of a thread's context
- This should NOT be called during system initialization

NetSilicon

Kernel Services Summary

- Kernel Service APIs
  - tx_time_set() / tx_time_get()
  - tx_interrupt_control()

Summary

Bringing up the ThreadX Kernel …

- … is quite easy
- Assumes a properly initialized C runtime environment (this is usually the case when entering main())
- Starting the ThreadX Kernel only needs a call to tx_kernel_enter()
- Besides that a function of name tx_application_define() must be declared
- tx_application_define is called from within the ThreadX library

NetSilicon

Bringing up the ThreadX Kernel (cont.)

tx_application_define() example implementation:

```
void tx_application_define(void *first_unused_memory)
{
    /* Setup ThreadX tick timer for a 1 ms tick period */
    SetupOsTimer(OS_TICKRATE);

    /* Create the root thread.  */
    tx_thread_create(&rootThreadCB,              /* control block for thread */
                     "rootThread",               /* thread name              */
                     rootThread,                 /* entry function           */
                     0,                          /* parameter                */
                     rootThreadStack,            /* start of stack           */
                     THREAD_STACK_SIZE,          /* size of stack            */
                     1,                          /* priority                 */
                     1,                          /* preemption threshold     */
                     1,                          /* time slice threshold     */
                     TX_AUTO_START);             /* start immediately        */
}
```

Bringing up the ThreadX Kernel (cont.)

Setting up the System Tick Timer (NS7520):

```
int SetupOsTimer( unsigned long ticksPerSecond )
{
    unsigned long tmp;

    if (ticksPerSecond < 1)
          return (OS_TIMER_EINVAL);

    *(unsigned long *)0xffb00010 = 0;           /* use IRQ not FIQ to interrupt */
    *(unsigned long *)0xffb00010 |= 0x08000000; /* use sys clock as clock source */

    tmp = (55000000 / ticksPerSecond) - 1;
    *(unsigned long *)0xffb00010 |= tmp;

    /*--- set handler ---*/
    SetIntHdlr(TIMER1_INT, _tx_timer_interrupt);

    *(unsigned long *)0xffb00030 |= 0x00000020; /* enable the interrupt in the  */
                                                /*   interrupt enable register. */
    *(unsigned long *)0xffb00010 |= 0x40000000; /* enable interrupt */
    *(unsigned long *)0xffb00010 |= 0x80000000; /* enable timer */

    return (OS_TIMER_ERR_OK);
}
```


NetSilicon