NXP Semiconductors
Application Notes

Document Number: AN12765
Rev.0, 03/2020

i.MX RT600 DSP Enablement

XOS use cases

1. Introduction

i.MX RT600 includes a DSP processor core which is Cadence Xtensa HiFi4 Audio DSP processor, running at frequencies of up to 600 MHz.

The XOS embedded kernel from Cadence is designed for efficient operation on embedded system built using the Xtensa architecture. Although various parts of XOS continue to be tuned for efficient performance on the Xtensa hardware, most of the code is written in standard C and is not Xtensa-specific.

XOS provides for core system and thread management operations and a list of modules supported. In this application note, we give some examples to show how to use below modules and what need to take care in the special cases.

- Condition
- Event
- Interrupt
- Semaphore
- Message Queue

Contents

2. Abbreviations

This chapter provides an overview of the abbreviations as used in this document.

Abbreviations	Description
Tensilica	It is a part of Cadence Design Systems
TIE	Tensilica Instruction Extension
XOS	Xtensa Embedded OS
Xplorer	Integrated Development Environment, based on Eclipse platform
Xtensa	i.MXRT600 DSP is based on Xtensa architecture

3. The XOS System Module

XOS is built as a library and statically linked to the application code to generate a single executable file. Make sure that the library is included in Xplorer if you want to use it.

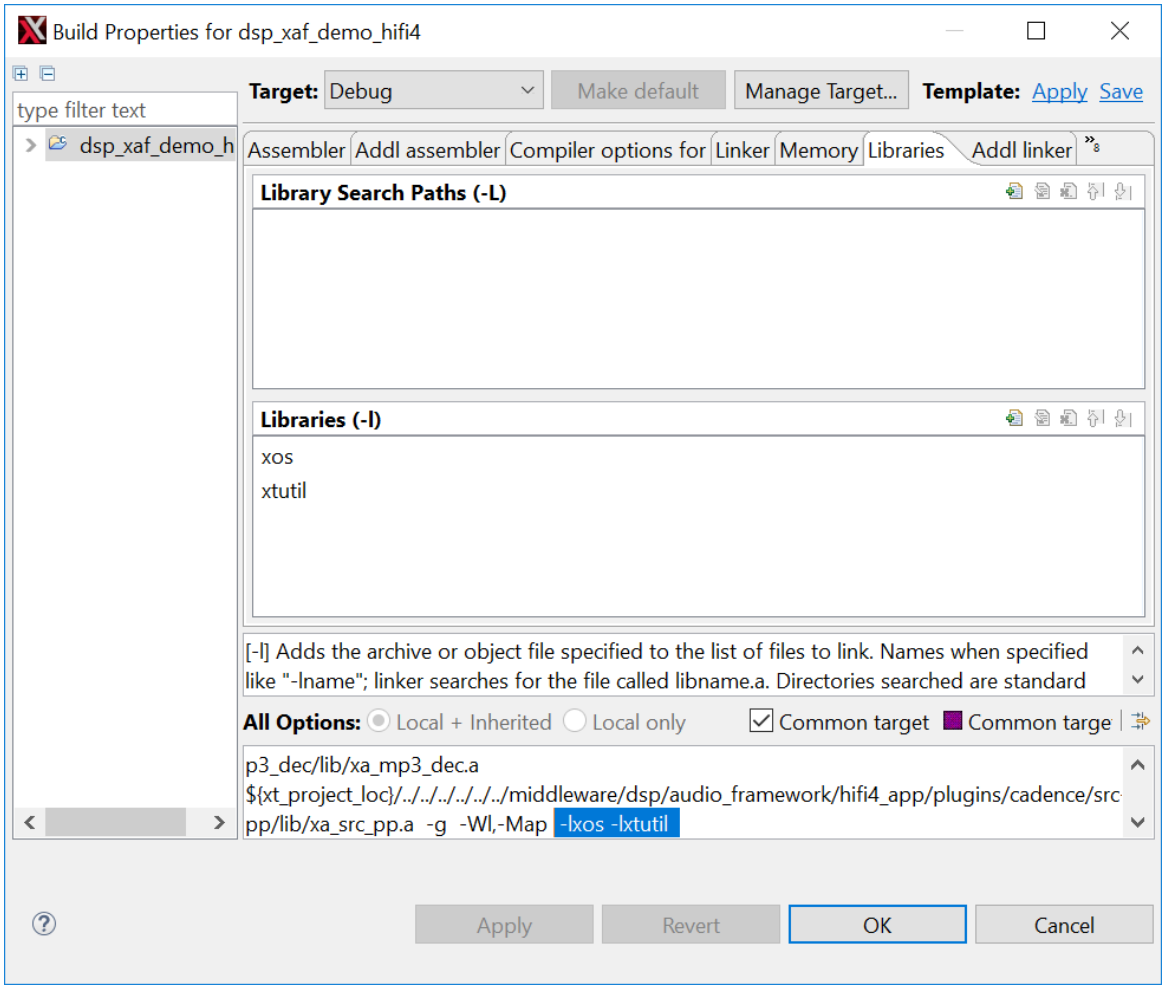


Figure 1. XOS library

XOS code runs at the same privilege level as the application code, and there is no protection for XOS data from accidental or malicious corruption by application code. XOS system calls are regular function calls, not traps. This makes execution faster and code size smaller.

Application threads can be created without the limitation of the number – this is limited only by memory available for thread control blocks and thread stacks.

Thread can exist in one of three possible states – Ready, Running, or Blocked. When a thread is created, it is put into either the Ready state or the Blocked state, depending on what was specified during creation.

- A thread in the Ready state is placed at the end of the ready queue for its priority level, and eventually will get to run, at which time it goes into the Running state.
- A Running thread may execute a blocking system call and go into the Blocked state. It can also be pre-empted by at thread at a higher priority and go back into the Ready state.
- A thread in the Blocked state does not go back into the Ready state until the blocking condition is satisfied.

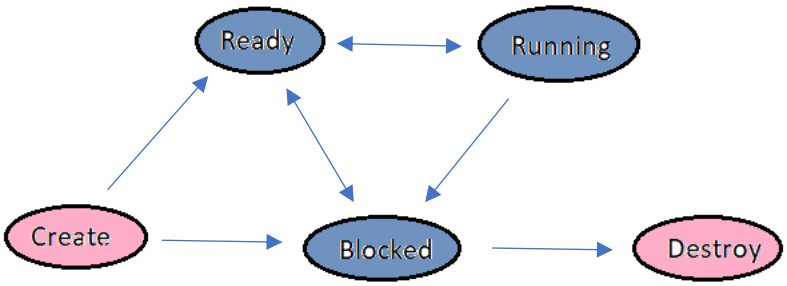


Figure 2. XOS Thread States

The brief description of thread functions as below

- xos_start() Init XOS and start multithreading.
- xos_start_main() Init XOS and convert main() into a thread.
- xos_thread_create() Create a new thread.
- xos_threadp_set_cp_mask() Set the coprocessor mask for a thread, in thread creation parameters.
- xos_threadp_set_preemption_priority() Set the preemption priority for a thread, in thread creation parameters.
- xos_threadp_set_exit_handler() Set the exit handler function for a thread, in thread creation parameters.
- xos_thread_delete() Destroy the thread, free up resources.
- xos_thread_abort() Force the thread to terminate.
- xos_thread_exit() Exit the current thread.
- xos_thread_join() Wait for a specified thread to terminate.
- xos_thread_yield() Yield the CPU.
- xos_thread_suspend() Suspend the specified thread.
- xos_thread_resume() Resume (make ready) the specified thread.
- xos_thread_get_priority() Get current priority of thread.
- xos_thread_set_priority() Set current priority of thread.

- xos_thread_set_exit_handler() Set exit handler function for thread.
- xos_thread_id() Return ID (handle) of current thread.
- xos_thread_get_name() Return thread name.
- xos_thread_set_name() Set thread name.
- xos_thread_cp_mask() Get list of CPs that this thread can touch.
- xos_thread_get_wake_value() Get last wake value for the thread.
- xos_thread_get_event_bits() Get current value of event bits for the thread.
- xos_thread_get_state() Get the state of the thread.
- xos_preemption_disable() Disable thread preemption.
- xos_preemption_enable() Re-enable thread preemption.
- xos_thread_get_stats() Get runtime statistics for the thread.
- xos_get_cpu_load() Calculate and return the CPU use % for all threads in the system.

Note: Thread stack is allocated by the caller. The size at least should be able to save coprocessor state, non-coprocessor TIE state and allocate an interrupt/exception frame plus whatever the thread actually needs. XOS supports multiple priority levels is configurable at build time, and zero is the lowest priority level.

4. Condition

Condition objects (or condition variables) allow thread to block and wait for a specific condition to become true. The condition is evaluated by a supplied condition function, and the evaluation is performed every time the condition object is signaled by another thread or by an interrupt handler.

The brief description of condition functions as below

- xos_cond_create() Initialize the condition object.
- xos_cond_delete() Destroy the condition object. Any waiting threads are unblocked.
- xos_cond_wait() Wait on the condition for it to become true.
- xos_cond_signal() Signal the condition, wake all waiting threads.
- xos_cond_signal_one() Signal the condition, wake one waiting thread.
- xos_cond_wait_mutex() Atomically release mutex and wait on condition.
- xos_cond_wait_mutex_timeout() As above, except a timeout can be specified for the wait.

In xos_condition example code, there are two condition objects initiated and two threads created. In the beginning of thread1_fun(), it needs to wait cond_1. When thread[0] runs thread0_fun() and signaled cond_1, it will wake up all threads waiting on this condition and also pass “value” to all waiters. So in thread1_fun() when ret got the default value 1234, then it can do ret+111 and wake up cond_0 then pass 1345 to waiter. This example is not only helping user to know how to use basic condition functions but also going to show how multiple threads wait/signal conditions and pass value to waiter. Below is code snippet and log for reference.

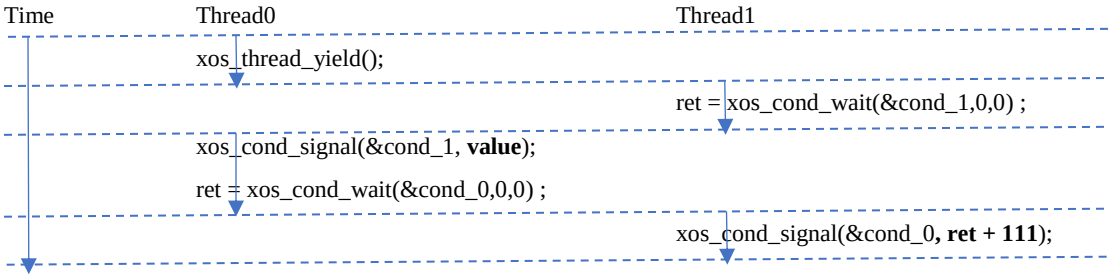
```
…
int thread0_fun(void * arg, int32_t unused)
{...
   xos_thread_yield();
   xos_cond_signal(&cond_1, value);
   ret = xos_cond_wait(&cond_0, 0, 0);
…}
int thread1_fun(void * arg, int32_t unused)
```

```
Log:
start XOS_COND_EX
thread0 created successfully
                thread1 created successfully
                thread_func():Thread0 starting
        thread_func():Thread1 starting
signal cond1
cond1 triggered and got ret=1234
```

```
{...
   ret = xos_cond_wait(&cond_1, 0, 0);
   xos_cond_signal(&cond_0, ret + 111);
...}
…
```

```
cond0 triggered and got ret=1345

terminate Thread0, code=0
terminate Thread1, code=0
XOS_COND_EX finished
```

Below flowchart also shows context switch between threads of xos_condition example.



5. Event

An event is a group of bits that can be set, cleared, and waited upon in various combinations. Events can be used for synchronization between threads, or between threads and interrupt handlers. XOS event objects can be both waited upon and signaled by multiple threads concurrently.
The brief description of event functions as below

- xos_event_create() Create the event object. Specify the group of valid bits.
- xos_event_delete() Destroy an event object. Will unblock all waiting threads.
- xos_event_set() Set the specified group of bits.
- xos_event_clear() Clear the specified group of bits.
- xos_event_clear_and_set() Clear one group and set another group, as one action. The groups may overlap.
- xos_event_get() Read the state of the event bits without blocking.
- xos_event_wait_all() Wait for all of a group of bits to be set.
- xos_event_wait_all_timeout() Like xos_event_wait_all(), except that a timeout can be specified for the wait.
- xos_event_wait_any() Wait for any of a group of events to be set.
- xos_event_wait_any_timeout() Like xos_event_wait_any(), except that a timeout can be specified for the wait.
- xos_event_set_and_wait() Atomically set a group of bits and wait on another group of bits.

In xos_event example code, there are three threads created. Thread0 first clear all bits and set bits 0-3 (0xf) then wait for bits 8-11 (0xf00). Thread1 runs case1 and wait for bits 0-3 but because bits 0-3 are set by thread0, it can move further then set bits 16-19. Thread2 runs case2 and bits 16-19 are set, so it does not need to wait and it can move further and set bits 8-11. At which time the waiting thread0 will be waked up. The purpose of this example is to show the interactions between multiple threads set the specified group of bits and wait for all of a group of bits to be set. Below is code snippet and log for reference.

```
…
int thread_func(void * arg, int32_t unused)
{...
   switch(flag) {
   case 0: // Clear all, set bits 0-3, wait on bits 8-11
```

```
Log:
start XOS_EVENT_EX
Thread0 ret = 0 created successfully
Thread1 ret = 0 created successfully
Thread2 ret = 0 created successfully
```

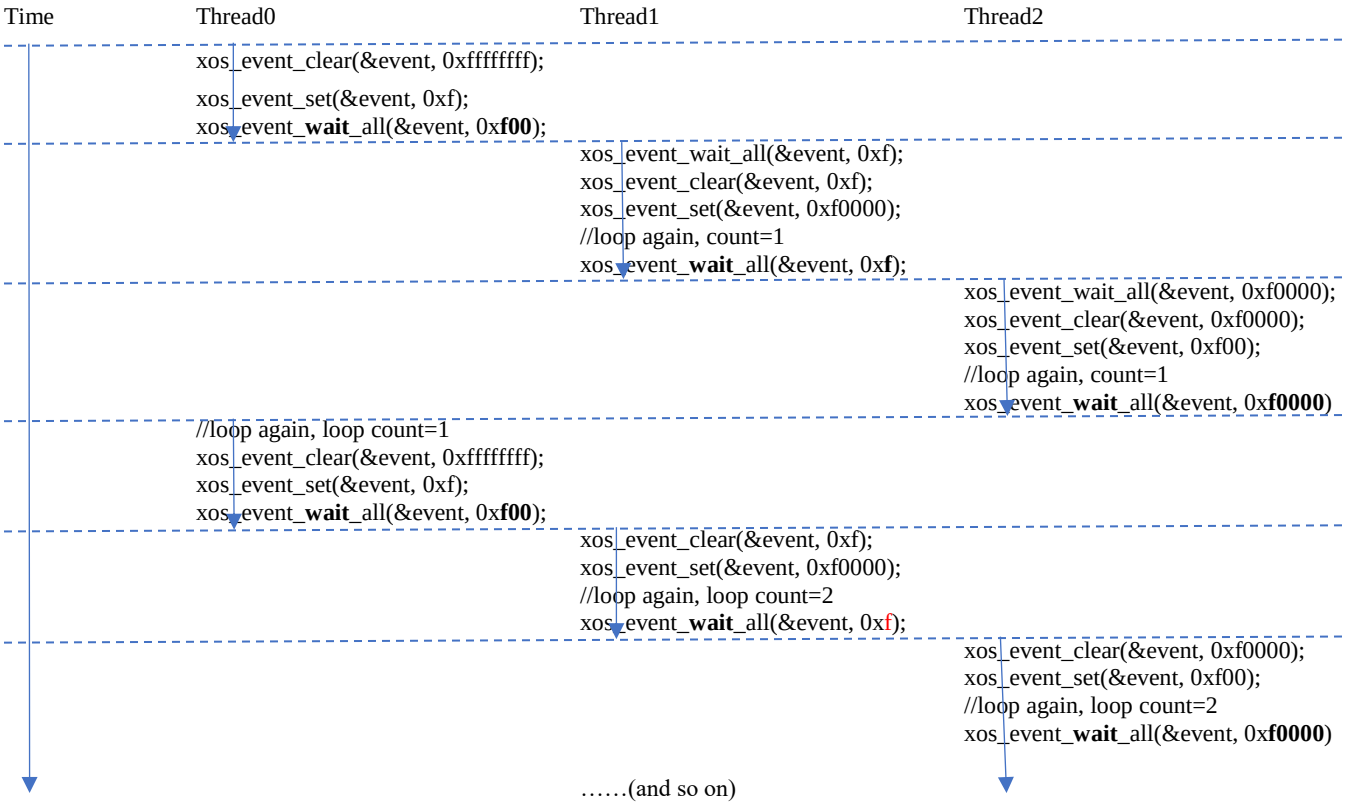
```
thread_func() cnt = 2 - Thread Thread2
thread_func() cnt = 2 - Thread Thread0
thread_func() cnt = 3 - Thread Thread1
thread_func() cnt = 3 - Thread Thread2
```

```
        xos_event_clear(&event, 0xffffffff);
        xos_event_set(&event, 0xf);
        xos_event_wait_all(&event, 0xf00);
    case 1: // Wait on bits 0-3, then set bits 16-19
        xos_event_wait_all(&event, 0xf);
        xos_event_clear(&event, 0xf);
        xos_event_set(&event, 0xf0000);
    case 2: // Wait on bits 16-19, then set bits 8-11
        xos_event_wait_all(&event, 0xf0000);
        xos_event_clear(&event, 0xf0000);
        xos_event_set(&event, 0xf00);
    }
}…
```

```
thread main prepare to do
thread_func():Thread0 starting
thread_func():Thread1 starting
thread_func() cnt = 1 - Thread Thread1

thread_func():Thread2 starting
thread_func() cnt = 1 - Thread Thread2

thread_func() cnt = 1 - Thread Thread0
thread_func() cnt = 2 - Thread Thread1
```

```
thread_func() cnt = 3 - Thread Thread0
thread_func() cnt = 4 - Thread Thread1
thread_func() cnt = 4 - Thread Thread2
thread_func() cnt = 4 - Thread Thread0
thread_func() cnt = 5 - Thread Thread1
thread_func() cnt = 5 - Thread Thread2
thread_func() cnt = 5 - Thread Thread0
terminate Thread0, code=5
terminate Thread1, code=5
terminate Thread2, code=5
XOS_EVENT_EX finished
```

Below flowchart also shows context switch between threads of xos_event example.



6. Timer Interrupt

The interrupt and exception dispatch mechanism in XOS was designed to make dispatching as fast as possible while still retaining flexibility for users to install their own customer handlers.
XOS supports nested interrupt handling. While an interrupt is being handled, if another interrupt at a higher priority level is asserted then the higher priority one is taken immediately, and it preempts the handling of the lower priority interrupt. If multiple interrupts at the same priority level are asserted at the same time, then the numerically highest one is handled first.
XOS uses a separate dedicated stack for interrupt processing. It switches to this stack when taking a interrupt handlers and functions called by these handlers use the interrupt stack, so the sizing of the interrupt stack is very import.

Threads have time-related services such as timed delays and timer callback functions. It provides the default preemption source, as the timer interrupt is used to trigger scheduling and time slicing.
The Xtensa processor can be configured with up to three internal timers. The system tick count is maintained as a 64-bit counter. All internal time tracking is done in terms of CPU clock cycles.
The brief description of time functions as below

- xos_set_clock_freq() Set system clock frequency.
- xos_get_clock_freq() Get current system clock frequency.
- xos_start_system_timer() Initialize timer support and start system timer.
- xos_get_system_timer_num() Get the ID of the system timer.
- xos_timer_init() Initialize a timer object.
- xos_timer_start() Start the timer object.
- xos_timer_stop() Stop the timer object.
- xos_timer_restart() Restart timer with new duration/period.
- xos_timer_is_active() Check if the timer is active.
- xos_timer_get_period() Get the repetition period for a periodic timer.
- xos_timer_set_period() Set the repetition period for a periodic timer.
- xos_get_system_cycles() Get current system cycle count.
- xos_thread_sleep() Suspend thread for specified number of CPU cycles.
- xos_thread_sleep_msec() Suspend thread for specified number of msec.
- xos_thread_sleep_usec() Suspend thread for specified number of usec.
- xos_timer_wait() Block calling thread on timer until it expires.
- gettimeofday() Get time and timezone information.
- settimeofday() Set time and timezone information.

In xos_interrupt example code, start the timer and call timer_fun() when the timer expires. By default, XOS uses the CCOUNT special register to count CPU clock cycles. The timer can be periodic (XOS_TIMER_PERIODIC) or the number of cycles from now(XOS_TIMER_DELTA). Below is code snippet and log for reference.

```
…
void timer_fun(void * arg)
{...
t2 = xos_get_ccount();
PRINTF("timer_fun(): t1=%u, t2=%u, diff is %u\r\n",
      (*t1), t2, (unsigned int)(t2-(*t1)));
}
int XOS_INTERRUPT_EX(void)
{...
   t1 = xos_get_ccount();

   // Set a timer for the near future.
   //xos_timer_start(&timer, delta, XOS_TIMER_DELTA,
1105037194
            timer_fun, (void*)&t1);
      xos_timer_start(&timer, delta, XOS_TIMER_PERIODIC,
            timer_fun, (void*)&t1);
...
}
```

```
Log:
start XOS_INTERRUPT_EX
timer_fun(): t1=2844683683, t2=3444684435, diff is 600000752
timer_fun(): t1=2844683683, t2=4044684901, diff is 1200001218

timer_fun(): t1=2844683683, t2=349718069, diff is 1800001682
timer_fun(): t1=2844683683, t2=949718539, diff is 2400002152
timer_fun(): t1=2844683683, t2=1549719007, diff is 3000002620
timer_fun(): t1=2844683683, t2=2149719475, diff is 3600003088
timer_fun(): t1=2844683683, t2=2749719943, diff is 4200003556
timer_fun(): t1=2844683683, t2=3349720411, diff is 505036728
            timer_fun(): t1=2844683683, t2=3949720877, diff is

...
```

7. Semaphore

Semaphores are construct that can be used to control access to a common resource from multiple threads. Semaphores have an associated count that controls the degree of access to the shared resource. XOS semaphores can be both waited upon and signaled by multiple threads concurrently.

The brief description of semaphore functions as below

- xos_sem_create() Create the semaphore and specify its properties.
- xos_sem_delete() Delete the semaphore. Will unblock all waiting threads.
- xos_sem_get() Decrement the semaphore count or block until able to do so.
- xos_sem_get_timeout() Like xos_sem_get(), except that a timeout can be specified for the wait.
- xos_sem_put() Signal the semaphore (increment count).
- xos_sem_put_max() Signal the semaphore only if the specified max count is not exceeded.
- xos_sem_tryget() Try to decrement the semaphore count but return immediately if failed.
- xos_sem_test() Check the value of the semaphore, but do not attempt to decrement it.

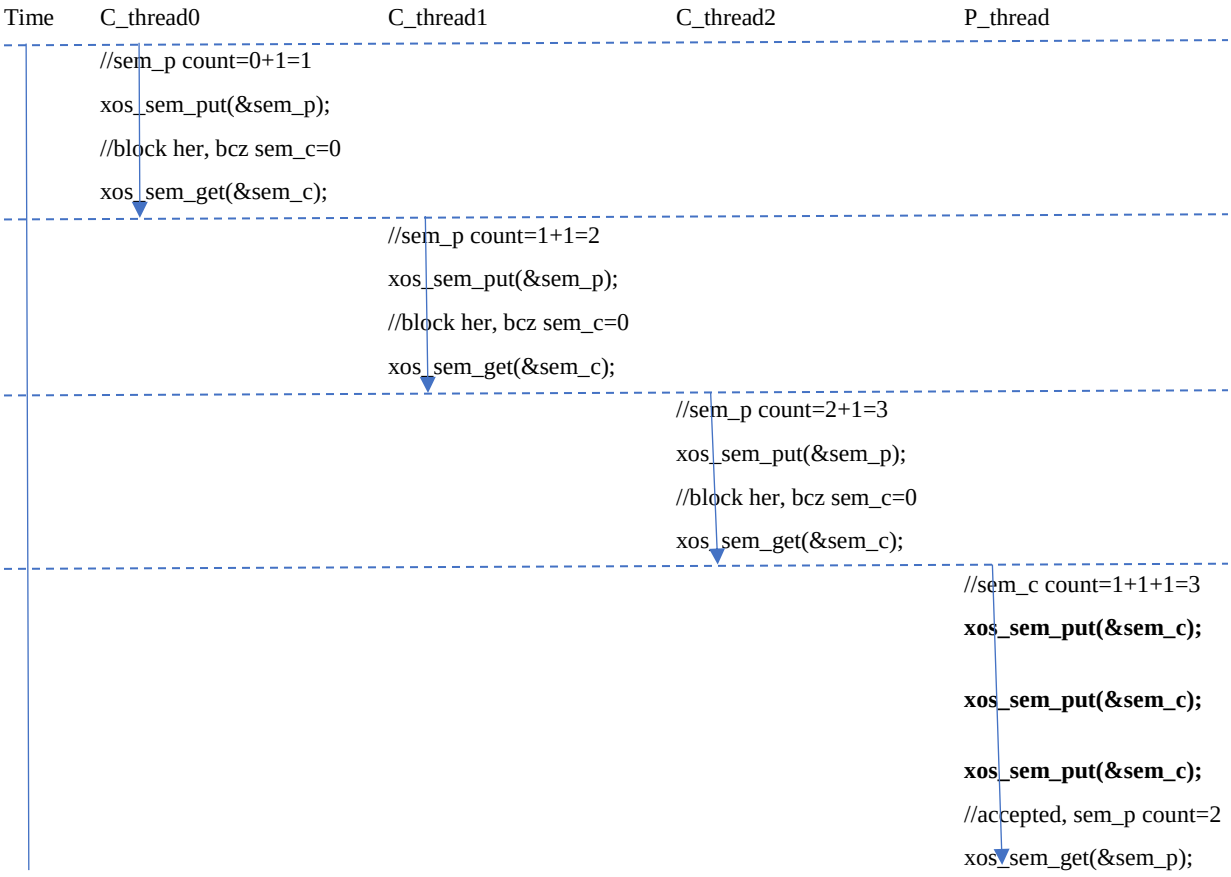
The well-known Producer-Consumer problem is a classic example of multiple threads to use semaphores for mutual exclusion and synchronization. The problem describes two threads the producer and the consumer, who generate data and put into the buffer (increase semaphore counter) and consume the data (decrease semaphore counter). When increasing the semaphore counter, it may wake up a waiting thread and if that thread is higher priority then there is an immediate context switch. But if decreasing the semaphore counter, it may block until the decrement is possible. In xos_semaphore example code, it provided an unbalanced producing and consuming so user can watch the changes of counter step by step. Below is code snippet and log for reference.

```
…
int32_t consumer_thread(void * arg, int32_t unused)
{...
   {//Signal the semaphore (increment count)
      ret = xos_sem_put(&semaphore_producer);

      //Decrement the semaphore count or block until able to do so
      ret = xos_sem_get(&semaphore_consumer);
   }...
}
int32_t producer_thread(void * arg, int32_t unused)
{...
   {//Signal 3 times just want to observe the behavior
      ret = xos_sem_put(&semaphore_consumer);
      ret = xos_sem_put(&semaphore_consumer);
      ret = xos_sem_put(&semaphore_consumer);
      //Decrement the semaphore count or block until able to do so
      ret = xos_sem_get(&semaphore_producer);
   ...
}
```

```
Log:
start XOS_SEM_EX
consumer_thread0 ret = 0 created successfully
consumer_thread1 ret = 0 created successfully
consumer_thread2 ret = 0 created successfully
producer_thread created successfully
thread_func():consumer_thread0 starting
consumer_thread0: semaphore_producer count = 1
thread_func():consumer_thread1 starting
consumer_thread1: semaphore_producer count = 2
thread_func():consumer_thread2 starting
consumer_thread2: semaphore_producer count = 3
thread_func():producer_thread starting
producer_thread: semaphore_consumer count = 3
producer_thread: Producer accepted item.
consumer_thread0: Consumer accepted item.
consumer_thread1: Consumer accepted item.
consumer_thread2: Consumer accepted item.
producer_thread: semaphore_consumer count = 3
producer_thread: Producer accepted item.
consumer_thread0: semaphore_producer count = 2
consumer_thread0: Consumer accepted item.
consumer_thread1: semaphore_producer count = 3
consumer_thread1: Consumer accepted item.
consumer_thread2: semaphore_producer count = 4
consumer_thread2: Consumer accepted item.
producer_thread: semaphore_consumer count = 3
producer_thread: Producer accepted item.
consumer_thread0: semaphore_producer count = 4
```

```
consumer_thread0: Consumer accepted item.
consumer_thread1: semaphore_producer count = 5
consumer_thread1: Consumer accepted item.
consumer_thread2: semaphore_producer count = 6
consumer_thread2: Consumer accepted item.
producer_thread: semaphore_consumer count = 3
producer_thread: Producer accepted item.
consumer_thread0: semaphore_producer count = 6
consumer_thread0: Consumer accepted item.
consumer_thread1: semaphore_producer count = 7
consumer_thread1: Consumer accepted item.
consumer_thread2: semaphore_producer count = 8
consumer_thread2: Consumer accepted item.
producer_thread: semaphore_consumer count = 3
producer_thread: Producer accepted item.
consumer_thread0: semaphore_producer count = 8
consumer_thread0: Consumer accepted item.
consumer_thread1: semaphore_producer count = 9
consumer_thread1: Consumer accepted item.
consumer_thread2: semaphore_producer count = 10
consumer_thread2: Consumer accepted item.
terminate consumer_thread0, code=0
terminate consumer_thread1, code=0
terminate consumer_thread2, code=0
terminate producer_thread, code=0
XOS_SEM_EX finished
```

Below flowchart also shows context switch between threads of xos_semaphore example. Note: consumer is abbreviated as ‘c”, producer is abbreviated as ‘p’ and semaphore is abbreviated as ‘sem’.



8. Message Queue

The message queue module implements a multi-writer multi-reader queue. It is thread-safe and can be used by interrupt handlers. Messages are copied into the queue so the message can be freed or reused as soon as the API calls return. The queue contains storage for a fixed number of messages, this number being defined at queue creation time.

The brief description of message queue functions as below

- xos_msgq_create() Initialize a message queue.
- xos_msgq_delete() Delete a message queue.
- xos_msgq_put() Put a message into the specified queue, wait until space is available.
- xos_msgq_put_timeout() Like xos_msgq_put(), except that a timeout can be specified for the wait.
- xos_msgq_get() Get a message from the specified queue, wait until a message is available.
- xos_msgq_get_timeout() Like xos_msgq_get(), except that a timeout can be specified for the wait.
- xos_msgq_empty() Check if the message queue is empty.
- xos_msgq_full() Check if the message queue is full.

In xos_queue example code, if PUTGET_BALANCE defined, it is very straightforward to observe put and get message sequentially. Three threads are created to put messages into the queue and three threads are created to get message from the queue. However, if PUTGET_BALANCE undefined, it shows that put message is faster than get message, because it put 3 messages but only get a message in one round.

NOTE

In this example, it tries to put message 1~10 to queue. Once message queue (ex: max data number is 10) is full, then thread waits here until space is available.

Below is code snippet and log for reference.

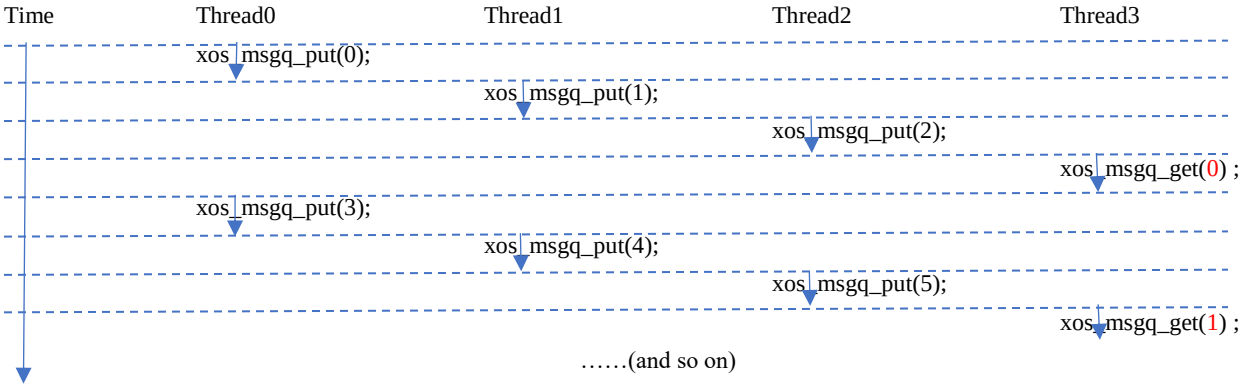
```
int put_func(void * arg, int32_t unused)
{...
      ret = xos_msgq_put(msgqueue, (uint32_t *)psrc);
}

int get_func(void * arg, int32_t unused)
{...
      ret = xos_msgq_get(msgqueue, (uint32_t *)&recv);
}

int XOS_QUEUE_EX(void)
{...
   ret = xos_thread_create(&thread[0], 0, put_func, 0,
   ret = xos_thread_create(&thread[1], 0, put_func, 0,
   ret = xos_thread_create(&thread[2], 0, put_func, 0,

   ret = xos_thread_create(&thread[3], 0, get_func, 0,
#if 0 //PUTGET_BALANCE
   ret = xos_thread_create(&thread[3], 0, get_func, 0,
   ret = xos_thread_create(&thread[3], 0, get_func, 0,
#endif
...
}
```

```
Log:
start XOS_QUEUE_EX
put_thread0: put_idx=0
put 0
put_thread1: put_idx=1
put 1
put_thread2: put_idx=2
put 2
get_thread0: get_idx=0
get 0
put_thread0: put_idx=3
put 3
put_thread1: put_idx=4
put 4
put_thread2: put_idx=5
put 5
get_thread0: get_idx=1
get 1
put_thread0: put_idx=6
put 6
put_thread1: put_idx=7
put 7
put_thread2: put_idx=8
put 8
get_thread0: get_idx=2
get 2
put_thread0: put_idx=9
put 9
put_thread1: put_idx=10
put 10
put_thread2: put_idx=11
put 11
get_thread0: get_idx=3
```

```
Message Queue is full,
   thread will wait here
   until space is available
get_thread0: get_idx=5
get 5
put_thread0: put_idx=17
put 17
get_thread0: get_idx=6
get 6
put_thread0: put_idx=18
put 18
get_thread0: get_idx=7
get 7
put_thread0: put_idx=19
put 19
get_thread0: get_idx=8
get 8
terminate put_thread0, code=0
put 14
get_thread0: get_idx=9
get 9
put 16
get_thread0: get_idx=10
get 10
terminate put_thread1, code=0
terminate put_thread2, code=0
get_thread0: get_idx=11
get 11
get_thread0: get_idx=12
get 12
get_thread0: get_idx=13
get 13
```

```
get 3                                   get_thread0: get_idx=14
put_thread0: put_idx=12                 get 15
put 12                                  get_thread0: get_idx=15
put_thread1: put_idx=13                 get 17
put 13                                  get_thread0: get_idx=16
put_thread2: put_idx=14                 get 18
Message Queue is full,                  get_thread0: get_idx=17
      thread will wait here             get 19
   until space is available             get_thread0: get_idx=18
get_thread0: get_idx=4                  get 14
get 4                                   get_thread0: get_idx=19
put_thread0: put_idx=15                 get 16
put 15                                  terminate get_thread0, code=0
put_thread1: put_idx=16
```

Below flowchart also shows context switch between threads of xos_queue example. Note: While creating the message queue object, memory and size for the queue must be allocated by the caller, either statically or via dynamic allocation.



9. XOS Initialization Sample Code

Here are two small examples to illustrate XOS initialization and startup.

The first one illustrates the case when main() is not converted into a thread.

```
#define STACK_SIZE (XOS_STACK_MIN_SIZE + 0x1000)
XosThread thread_tcb;
uint8_t thread_stack[STACK_SIZE];
int32_t thread_func(void * arg, int32_t unused)
{
        int32_t count = 0;
        puts("Thread starting.");
        while (1) {
                xos_thread_sleep(1000);
                count++;
                printf("Count = %dnn", count);
```

```
        }
        return 0;
}

int main()
{
        int32_t ret;
        // Set clock frequency before calling xos_start().
        xos_set_clock_freq(XOS_CLOCK_FREQ);
        // Select and start system timer.
        xos_start_system_timer(-1, 0);
        // Create at least one thread before calling xos_start().
        ret = xos_thread_create(&thread_tcb, 0, thread_func, 0, "demo", thread_stack, STACK_SIZE, 7, 0, 0);
        // Start multitasking.
        xos_start(0);
        // Should never get here.
        return -1;
}
```

Second example illustrates the case where main() is converted into a thread.

```
#define STACK_SIZE (XOS_STACK_MIN_SIZE + 0x1000)
XosThread thread_tcb;
uint8_t thread_stack[STACK_SIZE];
int32_t thread_func(void * arg, int32_t unused)
{
        int32_t count = 0;
        puts("Thread starting.");
        while (1) {
                xos_thread_sleep(1000);
                count++;
                printf("Count = %dnn", count);
        }
        return 0;
}

int main()
{
        int32_t ret;
```

```
    // Set clock frequency before calling xos_start_main().
    xos_set_clock_freq(XOS_CLOCK_FREQ);
    // Select and start system timer.
    xos_start_system_timer(-1, 0);
    // Start multitasking.
    xos_start_main("main", 5, 0);
    // Create a thread after control returns.
    ret = xos_thread_create(&thread_tcb, 0, thread_func, 0, "demo", thread_stack, STACK_SIZE, 7, 0, 0);
    // Do not return from here.
    while (1);
    return 0;
}
```

10. References

1. i.MXRT 600 Data Sheet
2. i.MXRT 600 User Manual (UM11147)
3. Xtensa Software Development Toolkit User's Guide
4. Xtensa XOS Reference Manual
5. Xtensa System Software Reference Manual

11. Revision History

Revision number	Date	Substantive changes
0	03/2020	Initial release

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN12765
Rev.0
03/202020