NXP Semiconductors Application Notes

# 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

1.	Introduction	1
2.	Abbreviations	2
3.	The XOS System Module	2
4.	Condition	
5.	Event	5
6.	Timer Interrupt	6
7.	Semaphore	8
8.	Message Queue	10
	XOS Initialization Sample Code	
10.	References	14
11.	Revision History	14



# 2. Abbreviations

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

Description

Abbreviations

- 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.

Build Properties for	dsp_xaf_demo_hifi4		×
	Target:     Debug     Make default     Manage Target     Temple	ate: <u>Apply</u>	<u>Save</u>
> 😂 dsp_xaf_demo_h	Assembler Addl assembler Compiler options for Linker Memory Libraries A	مddl linker <sup>»</sup>	3
	Library Search Paths (-L)	🗿 🗟 🗐 🖗	지 문 [
	Libraries (-l)	🗐 🗟 🔊 🖗	a (g)
	xos		
	xtutil		
	[-l] Adds the archive or object file specified to the list of files to link. Names whe	en specified	^
	like "-Iname"; linker searches for the file called libname.a. Directories searched a		~
	All Options: O Local + Inherited O Local only Common target	ommon targ	je 🗍 齢
	p3_dec/lib/xa_mp3_dec.a		^
< >	<pre>\${xt_project_loc}/////middleware/dsp/audio_framework/hifi4_app/plugii pp/lib/xa_src_pp.a -g -WI,-Map -lxos -lxtutil</pre>	ns/cadence/s	src <sup>.</sup>
?	Apply Revert OK	Cancel	
$\odot$	Appy Nevert OK	Cancel	

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.

### Condition

- 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 signal cond1 cond1 triggered and got ret=1234

```
Event
```

```
{...
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.

```
        Time
        Thread0
        Thread1

        xos_thread_yield();
        ret = xos_cond_wait(&cond_1,0,0);

        xos_cond_signal(&cond_1, value);
        ret = xos_cond_wait(&cond_0,0,0);

        ret = xos_cond_wait(&cond_0,0,0);
        xos_cond_signal(&cond_0, ret + 111);
```

# 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.

	Log:	
int thread_func(void * arg, int32_t unused)	start XOS_EVENT_EX	thread_func() cnt = 2 - Thread Thread2
{	Thread0 ret = 0 created successfully	thread_func() cnt = 2 - Thread Thread0
switch(flag) {	Thread1 ret = 0 created successfully	thread_func() cnt = $3$ - Thread Thread1
case 0: // Clear all, set bits 0-3, wait on bits 8-11	Thread2 ret = 0 created successfully	thread_func() cnt = $3$ - Thread Thread2

#### **Timer Interrupt**

<pre>xos_event_clear(&amp;event, 0xffffffff);</pre>	thread main prepare to do	thread_func() cnt = 3 - Thread Thread0
<pre>xos_event_set(&amp;event, 0xf);</pre>	thread_func():Thread0 starting	thread_func() cnt = 4 - Thread Thread1
xos_event_wait_all(&event, 0xf00);	thread_func():Thread1 starting	thread_func() cnt = 4 - Thread Thread2
case 1: // Wait on bits 0-3, then set bits 16-19	thread_func() $cnt = 1$ - Thread Thread1	thread_func() cnt = 4 - Thread Thread0
<pre>xos_event_wait_all(&amp;event, 0xf);</pre>		thread_func() cnt = 5 - Thread Thread1
xos_event_clear(&event, 0xf);	thread_func():Thread2 starting	thread_func() cnt = 5 - Thread Thread2
xos_event_set(&event, 0xf0000);	thread_func() $cnt = 1$ - Thread Thread2	thread_func() cnt = 5 - Thread Thread0
case 2: // Wait on bits 16-19, then set bits 8-11		terminate Thread0, code=5
xos_event_wait_all(&event, 0xf0000);	thread_func() cnt = 1 - Thread Thread0	terminate Thread1, code=5
xos_event_clear(&event, 0xf0000);	thread_func() cnt = $2$ - Thread Thread 1	terminate Thread2, code=5
<pre>xos_event_set(&amp;event, 0xf00);</pre>		XOS_EVENT_EX finished
}		
}		

Below flowchart also shows context switch between threads of x	os_event example.
--	-------------------

Time	Thread0	Thread1	Thread2
	xos_event_clear(&event, 0xffffffff);		
	xos_event_set(&event, 0xf); xos_event_wait_all(&event, 0xf00);		
		xos_event_wait_all(&event, 0xf);	
		xos_event_clear(&event, 0xf); xos_event_set(&event, 0xf0000);	
		//loop again, count=1	
		xos_event_wait_all(&event, 0xf);	
			<pre>xos_event_wait_all(&amp;event, 0xf0000); xos_event_clear(&amp;event, 0xf0000); xos_event_set(&amp;event, 0xf00); //loop again, count=1 xos_event_wait_all(&amp;event, 0xf0000)</pre>
	//loop again, loop count=1		
	<pre>xos_event_clear(&amp;event, 0xffffffff); xos_event_set(&amp;event, 0xf);</pre>		
	xos_event_wait_all(&event, 0xf00);		
		<pre>xos_event_clear(&amp;event, 0xf); xos_event_set(&amp;event, 0xf0000); //loop again, loop count=2 xos_event_wait_all(&amp;event, 0xf);</pre>	
			xos_event_clear(&event, 0xf0000); xos_event_set(&event, 0xf00); //loop again, loop count=2 xos_event_wait_all(&event, 0xf0000)
★		(and so on)	*

## 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.

i.MX RT600 DSP Enablement, Application Notes, Rev.0, 03/2020

#### **Timer Interrupt**

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)
                                                                 Log:
                                                                  start XOS_INTERRUPT_EX
{...
                                                                  timer_fun(): t1=2844683683, t2=3444684435, diff is 600000752
t2 = xos\_get\_ccount();
PRINTF("timer_fun(): t1=\%u, t2=\%u, diff is \%u\r\n",
                                                                  timer fun(): t1=2844683683, t2=4044684901, diff is 1200001218
    (*t1), t2, (unsigned int)(t2-(*t1)));
                                                                  timer_fun(): t1=2844683683, t2=349718069, diff is 1800001682
int XOS_INTERRUPT_EX(void)
                                                                  timer fun(): t1=2844683683, t2=949718539, diff is 2400002152
                                                                  timer fun(): t1=2844683683, t2=1549719007, diff is 3000002620
{...
 t1 = xos\_get\_ccount();
                                                                  timer fun(): t1=2844683683, t2=2149719475, diff is 3600003088
                                                                  timer fun(): t1=2844683683, t2=2749719943, diff is 4200003556
  // Set a timer for the near future.
                                                                  timer_fun(): t1=2844683683, t2=3349720411, diff is 505036728
  //xos_timer_start(&timer, delta, XOS_TIMER_DELTA,
                                                                           timer_fun(): t1=2844683683, t2=3949720877, diff is
1105037194
         timer_fun, (void*)&t1);
                                                                  ...
    xos_timer_start(&timer, delta, XOS_TIMER_PERIODIC,
         timer_fun, (void*)&t1);
```

```
}
```

### 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.

```
Log:
int32_t consumer_thread(void * arg, int32_t unused)
                                                                  start XOS_SEM_EX
                                                                 consumer_thread0 ret = 0 created successfully
{...
  {//Signal the semaphore (increment count)
                                                                 consumer_thread1 ret = 0 created successfully
    ret = xos_sem_put(&semaphore_producer);
                                                                 consumer_thread2 ret = 0 created successfully
                                                                  producer_thread created successfully
    //Decrement the semaphore count or block until able to do so
                                                                 thread func():consumer thread0 starting
    ret = xos_sem_get(&semaphore_consumer);
                                                                  consumer_thread0: semaphore_producer count = 1
                                                                  thread_func():consumer_thread1 starting
  }...
                                                                 consumer_thread1: semaphore_producer count = 2
int32_t producer_thread(void * arg, int32_t unused)
                                                                  thread_func():consumer_thread2 starting
                                                                 consumer_thread2: semaphore_producer count = 3
{...
  {//Signal 3 times just want to observe the behavior
                                                                  thread_func():producer_thread starting
    ret = xos_sem_put(&semaphore_consumer);
                                                                  producer_thread: semaphore_consumer count = 3
                                                                 producer_thread: Producer accepted item.
    ret = xos_sem_put(&semaphore_consumer);
    ret = xos_sem_put(&semaphore_consumer);
                                                                 consumer_thread0: Consumer accepted item.
    //Decrement the semaphore count or block until able to do so
                                                                 consumer thread1: Consumer accepted item.
    ret = xos_sem_get(&semaphore_producer);
                                                                 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
```

#### Semaphore

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 = 3producer\_thread: Producer accepted item. consumer\_thread0: semaphore\_producer count = 6consumer\_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'.



Message Queue



### 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)	Log:	
{	start XOS_QUEUE_EX	Message Queue is full,
ret = xos_msgq_put(msgqueue, (uint32_t *)psrc);	put_thread0: put_idx=0	thread will wait here
}	put 0	until space is available
	put_thread1: put_idx=1	get_thread0: get_idx=5
int get_func(void * arg, int32_t unused)	put 1	get 5
{	put_thread2: put_idx=2	put_thread0: put_idx=17
ret = xos_msgq_get(msgqueue, (uint32_t *)&recv);	put 2	put 17
}	get_thread0: get_idx=0	get_thread0: get_idx=6
	get 0	get 6
int XOS_QUEUE_EX(void)	put_thread0: put_idx=3	put_thread0: put_idx=18
{	put 3	put 18
ret = xos_thread_create(&thread[0], 0, put_func, 0,	put_thread1: put_idx=4	get_thread0: get_idx=7
ret = xos_thread_create(&thread[1], 0, put_func, 0,	put 4	get 7
ret = xos_thread_create(&thread[2], 0, put_func, 0,	put_thread2: put_idx=5	put_thread0: put_idx=19
	put 5	put 19
ret = xos_thread_create(&thread[3], 0, get_func, 0,	get_thread0: get_idx=1	get_thread0: get_idx=8
#if 0 //PUTGET_BALANCE	get 1	get 8
ret = xos_thread_create(&thread[3], 0, get_func, 0,	put_thread0: put_idx=6	terminate put_thread0, code=0
ret = xos_thread_create(&thread[3], 0, get_func, 0,	put 6	put 14
#endif	put_thread1: put_idx=7	get_thread0: get_idx=9
	put 7	get 9
}	put_thread2: put_idx=8	put 16
	put 8	get_thread0: get_idx=10
	get_thread0: get_idx=2	get 10
	get 2	terminate put_thread1, code=0
	put_thread0: put_idx=9	terminate put_thread2, code=0
	put 9	get_thread0: get_idx=11
	put_thread1: put_idx=10	get 11
	put 10	get_thread0: get_idx=12
	put_thread2: put_idx=11	get 12
	put 11	get_thread0: get_idx=13
	get_thread0: get_idx=3	get 13

#### **XOS Initialization Sample Code**

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.

Time	Thread0	Thread1	Thread2	Thread3
	xos msgq_put(0);			
		xos msgq_put(1);		
			xos_msgq_put(2);	
				xos_msgq_get(0);
	xos_msgq_put(3);			
		xos_msgq_put(4);		
			xos_msgq_put(5);	
				<pre>xos_msgq_get(1);</pre>
<b>↓</b>		(and so on)		

### 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 \text{ count} = 0;
         puts("Thread starting.");
         while (1) {
                   xos_thread_sleep(1000);
                   count++;
                   printf("Count = %dnn", count);
         }
         return 0;
}
int main()
{
```

int32\_t ret;

**Revision History** 

```
// 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

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive 3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, AMBA, Arm Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and uVision are registered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. Arm7, Arm9, Arm11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, Mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2020 NXP B.V.



i.MX RT600 DSP Enablement, Application Notes, Rev.0, 03/2020

#### How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support