Threads & Semaphores

# Introduction

This application note provides a brief about Threads and Semaphore APIs available in the SDK. Accompanying sample application describes the use of these APIs.

# Threads and Semaphores

## Threads

A thread is represented by the struct os\_thread. This object contains all information needed by the kernel to manage the execution of the thread.

Each thread is created with a stack and an entry point. The entry point is a function that the threads begin to execute when started by the kernel. The thread will continue to run until it returns from this function.

The threads are scheduled on a strict priority-based scheme, with eight different priority levels 0-7, 7 having the highest priority. The scheduler will preempt the currently running thread at any point if a thread with a higher priority becomes ready to execute.

## Semaphores

A semaphore provides a synchronized counter for a shared resource for use in a thread context. Each semaphore has a value (a non-negative number) representing *how much* of the resource is available. If the value is only ever zero or one, it can be used as a sleeping lock. Since waiting for a semaphore depends on the ability to sleep, it must not be done in an interrupt context.

# Threads and Semaphores APIs

## Thread handling APIs

### os\_create\_thread()

Creates a new thread with priority specified in flags.

|  |
| --- |
| struct os\_thread \* os\_create\_thread(const char \*name, os\_entrypoint\_t entry, os\_threadarg\_t arg, uint32\_t flags, size\_t stacksz) |

Though the thread is placed in the run queue, there is no immediate reschedule.

Parameters passed to the API are:

1. Name of the thread
2. Entry point for the thread
3. Argument passed to entry point - specifies attributes like a priority for the new thread
4. Requested stack size in bytes.

The thread continues to run until the entry point returns, at which point return value (a pointer) can be obtained with os\_join\_thread().

Returns pointer to struct os\_thread if successful, NULL otherwise.

### os\_join\_thread ()

Waits for a thread to terminate and destroy the thread.

|  |
| --- |
| void \* os\_join\_thread(struct os\_thread \*thread) |

Calling this function will suspend the execution of the calling thread until the target thread exits. The memory used to hold the threads stack and control block is freed.

The function returns the value returned by the terminating thread.

If the return value is of no consequence, OS\_CRTHREAD\_DETACHED can be passed in flags. This causes the OS to reap the thread.

### os\_self()

Return reference to the calling thread.

|  |
| --- |
| struct os\_thread \* os\_self(void) |

This function returns the reference to the calling thread. This is the same value that was returned from the os\_create\_thread() call used to create this thread.

## Semaphore handling APIs

### os\_sem\_init()

Initializes a semaphore when passed a pointer to the semaphore where the initial value is to be assigned to the semaphore.

|  |
| --- |
| void os\_sem\_init(struct os\_semaphore \*sem, int value) |

### os\_sem\_wait()

Locks a semaphore. If the value of the semaphore is greater than zero, it decrements the counter. If the value is zero, it puts the current thread to sleep until the value becomes positive.

|  |
| --- |
| void os\_sem\_wait(struct os\_semaphore \*sem) |

### os\_sem\_wait\_timeout()

Locks a semaphore with timeout passed as parameter tmo in microseconds.

|  |
| --- |
| int os\_sem\_wait\_timeout(struct os\_semaphore \*sem, uint32\_t tmo) |

If the value is zero, it puts the current thread to sleep for at most tmo microseconds. If the semaphore is unlocked before the timeout expires, it locks the semaphore and returns zero. If a timeout occurs, -1 is returned (and the semaphore is unlocked).

### os\_sem\_post()

Unlocks a semaphore. Increments the value of a semaphore and wakes the first thread waiting for this semaphore.

|  |
| --- |
| void os\_sem\_post(struct os\_semaphore \*sem) |

### os\_sem\_waiting()

Return true if there are any threads waiting for this semaphore.

|  |
| --- |
| bool os\_sem\_waiting(struct os\_semaphore \*sem) |

# Code Walkthrough

## Sample Application 1 – Thread Creation

**Note**: All the applicable ELFs are available in the following location of the SDK release package: sdk\_x.y\examples\innoos\_threads\_semaphores\bin).

x and y in sdk\_x.y refer to the SDK release version. For example: *sdk\_2.4\examples\innoos\_threads\_semaphores\bin*.

### Overview

The sample code in the path /examples/innoos\_threads\_semaphores /src/threads\_sample1.c provides more details on how to create a thread and execute it.

### Sample Code Walkthrough

We assign the priority and stack size of the thread using the preprocessing directives as follows:

|  |
| --- |
| #define MY\_APP\_THREAD\_PRIO 1 /\* thread priority\*/  #define MY\_APP\_THREAD\_STACK\_SIZE 512 /\* thread stack size\*/ |

After thread creation, we check if the creation of a thread was successful or not. If the call to os\_create\_thread() is successful, the OS schedules the thread to run where the function my\_app\_thread\_func() will be called. It prints the string output to the console and returns.

|  |
| --- |
| /\* creates a thread \*/  my\_app\_thread = os\_create\_thread("my\_app\_thread", my\_app\_thread\_func,NULL, MY\_APP\_THREAD\_PRIO, MY\_APP\_THREAD\_STACK\_SIZE); |

While the thread function is still running, the API os\_join\_thread()will suspend the main thread (calling thread) until the thread to be joined (i.e.. the ‘my\_app\_thread’ in this example) terminates.

|  |
| --- |
| if( my\_app\_thread ==NULL)  {  os\_printf(" thread creation failed\n");  return-1;  }  /\* waits for thread function to finish \*/  os\_join\_thread(my\_app\_thread); |

### Using of Delays in Threads

Delays are mainly used when we want to stop the execution flow of a program for t units of time. The libkernel.a library provides many function calls with different delay precision.

We can use the functions such as os\_msleep() for milliseconds, and os\_usleep() for microseconds.

i.e., os\_msleep(1000) will provide a 1000 millisecond delay. Consider the example where a delay function is used:

|  |
| --- |
| static void\* my\_app\_thread\_func(void\* arg)  {  os\_printf("1. hello world from a thread function\n");  os\_msleep(1000);  os\_printf("2. hello world after 1000ms\n");  returnNULL;  } |

When the thread function is called, it outputs the first string. Then the thread suspends for 1000 milliseconds and when it resumes execution and then prints the second string.

### Running the Application

Program threads\_sample1.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the threads\_sample1.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

threads\_sample1.elf is created when compiling this code which gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Threads Demo App 1  1. hello world from a thread function  2. hello world after 1000ms |

## Sample Application 2 – Thread Priorities

### Overview

The sample code in the path: /examples/innoos\_threads\_semaphores/src/threads\_sample2.c describes the creation of two threads with different priority levels.

### Sample Code Walkthrough

Priorities are assigned for each thread as follows:

|  |
| --- |
| #define MY\_APP\_THREAD\_PRIO 2 /\* thread priority ie, the first executing thread\*/  #define MY\_APP\_THREAD\_STACK\_SIZE 512 /\* thread stack size\*/  #define MY\_APP\_THREAD\_2\_PRIO 1 /\* thread priority ie, the second executing thread\*/  #define MY\_APP\_THREAD\_2\_STACK\_SIZE 512 /\* thread stack size\*/ |

Here, two thread variables are declared:

|  |
| --- |
| /\* declare os\_thread variables \*/  static struct os\_thread \*my\_app\_thread;  static struct os\_thread \*my\_app\_thread\_2; |

Post which, os\_create\_thread() API is called with the following boot arguments: name of the thread, entry point for the thread, argument for the entry point, thread priority flag and the stack size in bytes.

|  |
| --- |
| /\* create threads \*/  my\_app\_thread =os\_create\_thread("my\_app\_thread", my\_app\_thread\_func,NULL, MY\_APP\_THREAD\_PRIO, MY\_APP\_THREAD\_STACK\_SIZE);  if(NULL== my\_app\_thread)  {  os\_printf(" thread creation failed\n");  return-1;  }  my\_app\_thread\_2 =os\_create\_thread("my\_app\_thread\_2", my\_app\_thread\_func\_2,NULL, MY\_APP\_THREAD\_2\_PRIO, MY\_APP\_THREAD\_2\_STACK\_SIZE);  if(NULL== my\_app\_thread\_2 )  {  os\_printf(" thread creation failed\n");  return-1;  } |

Any type of variable can be passed as an argument to the threads. Here, the thread takes a string argument which is received as an output inside the thread function itself.

|  |
| --- |
| static void\* my\_app\_thread\_func\_2(void\* arg)  {  do  {  flag++;  os\_printf("Message from Second Thread. Thread id[%x]\n",(unsignedint)os\_self());  }while(flag<3);  returnNULL;  } |

To join threads, os\_join\_thread()API is used as shown in the following code block:

|  |
| --- |
| /\* waits for thread function to finish \*/  os\_join\_thread(my\_app\_thread);  os\_join\_thread(my\_app\_thread\_2); |

### Running the Application

Program threads\_sample2.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the threads\_sample2.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

threads\_sample2.elf is created when compiling this code which gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Threads Demo App 2  Message from First Thread. Thread id[bf780]  Message from First Thread. Thread id[bf780]  Message from First Thread. Thread id[bf780]  Message from Second Thread. Thread id[bfa00]  Message from Second Thread. Thread id[bfa00]  Message from Second Thread. Thread id[bfa00] |

## Sample Application 3 – Thread & Stack Size

### Overview

In the sample code in the path: /examples/innoos\_threads\_semaphores/src/threads\_sample3.c, threads are created with a given stack size in a loop. After creating a few threads, further thread creation fails due to the stack size. If the stack size for each thread is lowered, then multiple threads can co-exist.

### Sample Code Walkthrough

Thread stack sized defined as a macro:

|  |
| --- |
| #define THREAD\_BASICS\_STACK\_SIZE 1024 /\* thread stack size\*/ |

Here, threads are created in a loop until failure occurs. If thread creation fails, os\_create\_thread() returns a NULL pointer:

|  |
| --- |
| /\* creates a threads, until creation of thread fails \*/  while(1)  { thread\_id = os\_create\_thread("THREAD\_BASICS",thread\_sample\_thread\_func,  NULL, OS\_CRTHREAD\_DETACHED|THREAD\_BASICS\_PRIO, THREAD\_BASICS\_STACK\_SIZE);  if(NULL== thread\_id)  {  os\_printf("resource not available. thread creation failed.\n");  os\_error(OS\_ERR\_INTERNAL\_ERROR);  }  os\_sleep\_us(10\*1000, OS\_TIMEOUT\_NO\_WAKEUP);  } |

Each thread prints a text, and sleeps inside a while loop. If the thread is not sleeping, then thread creation might not fail. This is because, resources allocated for the earlier threads were on exit and are available for new threads.

|  |
| --- |
| /\* the thread function \*/  static void\* thread\_sample\_thread\_func(void\* arg)  {  os\_printf("%03d:thread created\n",++thread\_count);  while(1)  {    os\_sleep\_us(1000\*1000, OS\_TIMEOUT\_NO\_WAKEUP);  }  return NULL;  } |

Changing the stack size and the number of threads created before failure increases is as defined in the following code:

|  |
| --- |
| #define THREAD\_BASICS\_STACK\_SIZE 512 /\* thread stack size\*/ |

### Running the Application

Program threads\_sample3.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the threads\_sample3.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

threads\_sample3.elf is created when compiling this code which gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Threads Demo App 3  001:thread created  002:thread created  003:thread created  004:thread created  005:thread created  006:thread created  007:thread created  008:thread created  009:thread created  010:thread created  011:thread created  012:thread created  013:thread created  014:thread created  015:thread created  016:thread created  017:thread created  018:thread created  019:thread created  020:thread created  021:thread created  022:thread created  023:thread created  024:thread created  025:thread created  026:thread created  027:thread created.  .  .  .  .  .  .  .  697:thread created  698:thread created  resource not available. thread creation failed.  OS\_ERROR 0xfc  OS\_UNEXPECTED\_EXCEPTION 0x6  R0=00000030 R1=0004b6d4 R2=00fc0d00 R3=00000000  R4=00000200 R5=001049cd R6=00104a47 R7=07777777  R8=08888888 R9=09999999 R10=0aaaaaaa R11=0bbbbbbb  R12=000b2be3 SP=000b2c84 LR=00104a1b PC=00104a1a  xPSR=21000000 CONTROL=00000000 CFSR=00010000 BFAR=e000ed38  STACK:  0x000b2cc8: 00000200 01111111 04444444 05555555  0x000b2cd8: 06666666 00044e81 6e69616d 2988fa00  0x000b2ce8: 001049f1 07f83201 00000002 000b2c84  0x000b2cf8: 000b24e8 a5631209 000b2d00 000b2d00  0x000b2d08: 000b2d08 000b2d08 0004000c 0004000c  0x000b2d18: 000bf5e0 000bed20 000b2d20 000b2d20  0x000b2d28: 00000000 00000000 2df339e6 7349fc7f  0x000b2d38: 9ae964cd 00b1164b 345d406c 39c04a4e  0x000b2d48: ffc8daa2 abfbaff5 8fb90474 1b15e267  0x000b2d58: 7e319424 5ba8eb37 77759125 fdb7ba53  0x000b2d68: ebadcc99 201b8df4 0b444886 67873273  0x000b2d78: 297324ac 11501758 e51c63cf 036a56b6  0x000b2d88: 70db1ab6 040913ed aa791662 5dd9b7f3  0x000b2d98: e309dfe6 83476ca2 cd864e0a 33f21c84  0x000b2da8: 6c98c51e dcf15390 84ece867 1518ef2a  0x000b2db8: e50d55a8 b8202166 b3b87b4e 9974b048 |

## Sample Application 4 – Thread & Semaphores

### Overview

In the sample code in the path: /examples/innoos\_threads\_semaphores/src/threads\_semaphores.c, three threads are created, each with separate thread functions. By using a semaphore, the threads will execute according to the value used for the initialization of the semaphore variable.

In this application, Thread1 initially takes the semaphore while Thread2 and Thread3 wait until Thread1 completes its task and releases the semaphore. Thread2 and Thread3 are both waiting for the semaphore but in two different ways.

### Sample Code Walkthrough

Here we are using three threads as shown below:

|  |
| --- |
| my\_app\_thread\_1 =os\_create\_thread("my\_app\_thread\_1", my\_app\_thread\_func\_1, NULL, SEMAPHORE\_SAMPLE\_THREAD\_PRIORITY, SEMAPHORE\_SAMPLE\_THREAD\_STACK\_SIZE);  /\* creating thread 1 \*/  my\_app\_thread\_2 =os\_create\_thread("my\_app\_thread\_2",my\_app\_thread\_func\_2, NULL, SEMAPHORE\_SAMPLE\_THREAD\_PRIORITY, SEMAPHORE\_SAMPLE\_THREAD\_STACK\_SIZE);  /\* creating thread 2 \*/  my\_app\_thread\_3 =os\_create\_thread("my\_app\_thread\_3", my\_app\_thread\_func\_3,NULL, SEMAPHORE\_SAMPLE\_THREAD\_PRIORITY, SEMAPHORE\_SAMPLE\_THREAD\_STACK\_SIZE);  /\* creating thread 2 \*/ |

Before thread creation, there is a need to declare and initialize the semaphore variable by using the following statements:

|  |
| --- |
| static struct os\_semaphore my\_sem;  …  …  …  /\* initializes a semaphore. this has to be called before using my\_sem.  init with 0  \*/  os\_sem\_init(&my\_sem,0); |

It is required to initialize the semaphore variable using os\_sem\_init()before using the semaphore variable. In the code, the semaphore is initialized to 0.

Consider the thread functions defined for our created threads:

my\_app\_thread\_func\_1()for Thread1:

|  |
| --- |
| static void\* my\_app\_thread\_func\_1(void\* arg)  {  os\_printf("\n%x:%u:from thread 1. doing initial tasks",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*sleeping for some time, still other threads waiting\*/  os\_msleep(1000);  /\*checking is there any one waiting on the thread\*/  os\_printf("\n%x:%u:from thread 1. Is any one waiting for sem:%s",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(),  os\_sem\_waiting(&my\_sem)==1?"Yes":"No");  os\_printf("\n%x:%u:from thread 1. initial jobs done. now thread 2/3 can start."  " going to release the semaphore",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*releasing the semaphore\*/  os\_sem\_post(&my\_sem );    /\*checking is there any one waiting on the thread\*/  return NULL;  } |

my\_app\_thread\_func\_2() for Thread2.

Thread 2 is waiting to acquire the semaphore. Once it does, it waits for 3000 milliseconds post which it releases the semaphore.

|  |
| --- |
| static void\* my\_app\_thread\_func\_2(void\* arg)  {  os\_printf("\n%x:%u:from thread 2. waiting for semaphore",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*waiting on the semaphore\*/  os\_sem\_wait(&my\_sem );  os\_printf("\n%x:%u:from thread 2. got semaphore",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*thread 2 business logic here\*/  os\_printf("\n%x:%u:from thread 2. releasing semaphore\n",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*releasing the semaphore\*/  os\_sem\_post(&my\_sem );  return NULL;  } |

my\_app\_thread\_func\_3() for Thread3.

Thread 3 is waiting to acquire the semaphore until it times out. If it times out, it will loop back and once again try to acquire the semaphore.

Once it does, it will complete its task and releases the semaphore:

|  |
| --- |
| static void\* my\_app\_thread\_func\_3(void\* arg)  {  int ret;  os\_printf("\n%x:%u:from thread 3. waiting for semaphore",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  while(1)  {  /\*If the value of the semaphore is greater than zero,  decrement the counter and return zero; here the value of the semaphore is 1\*/  ret = os\_sem\_wait\_timeout(&my\_sem,1000000);  if(ret)  {  os\_printf("\n%x:%u:from thread 3. timedout. still waiting for semaphore. ret:%d",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(), ret);  /\*based on the application requirement, application can decide  whether to wait more or continue with 'could not lock' logic\*/  }  else  {  os\_printf("\n%x:%u:from thread 3. got semaphore. ret:%d",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(), ret);  break;  }  }  os\_printf("\n%x:%u:from thread 3. releasing semaphore\n",  SEMAPHORE\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*releasing the semaphore\*/  os\_sem\_post(&my\_sem );  returnNULL; } |

### Running the Application

Program threads\_semaphores.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the threads\_semaphores.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

threads\_semaphores.elf is created when compiling this code which gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Threads Semaphores Demo App  bfb80:95336:from thread 1. doing initial tasks  b35e8:95382:from thread 2. waiting for semaphore  b3868:95543:from thread 3. waiting for semaphore  bfb80:1095384:from thread 1. Is any one waiting for sem:Yes  bfb80:1095434:from thread 1. initial jobs done. now thread 2/3 can start. going to release the semaphore  b35e8:1095504:from thread 2. got semaphore  b3868:1095593:from thread 3. timedout. still waiting for semaphore. ret:-1  b3868:2095658:from thread 3. timedout. still waiting for semaphore. ret:-1  b3868:3095723:from thread 3. timedout. still waiting for semaphore. ret:-1  b35e8:4095550:from thread 2. releasing semaphore  b3868:4095601:from thread 3. got semaphore. ret:0  b3868:4095647:from thread 3. releasing semaphore |