Work Queues

# Introduction

This application note describes the concept and usage of Work Queues. Accompanying sample application provides a brief on how to use Work Queue APIs available with the SDK.

# Work Queue

Work Queues are used to defer a set of work by scheduling functions doing that work to run by a dedicated thread. For example, a set of work can be differed from interrupt context to thread context using this mechanism.

These deferred function calls are managed by an abstraction called work, defined by the struct os\_work, which is initialized with a work function by calling os\_init\_work().

A work queue is a list of os\_work objects which are initialized using os\_init\_workqueue().

Once the work queue is created and initialized, a set of Work objects can be Queued to this queue.

A dedicated thread called Worker Thread is created to service the work queue. This thread simply calls the function os\_run\_work() with a pointer to an initialized work queue as a single argument. Then the functions in the queued set of work are called from this thread’s context when the Worker Thread is given time to run by the scheduler.

More information on the APIs in the consecutive sections.

The system automatically creates a work queue on start-up that is used internally for executing system type work. For example, cleaning up after the threads have terminated. It is called a system work queue and can be used by applications having simple work functions, not requiring a dedicated thread context.

Alternatively, application defined threads can be used to service a work queue. This gives more control to the application to create many threads and manage their properties (stack size and priorities) as per the needs.

Two examples accompanying this application note showcase the system threads approach and application created threads approach for servicing the work queue.

# Work Queue APIs

## os\_init\_work()

Initializes a struct os\_work with a work function. Pointers to both the work functions are passed as input.

|  |
| --- |
| static inline void os\_init\_work(struct os\_work \*work, os\_workfn\_t func) |

## os\_init\_workqueue()

Initializes os\_workqueue struct.

|  |
| --- |
| void os\_init\_workqueue(struct os\_workqueue \*wq) |

## os\_run\_work()

This function is called to service a work queue.

|  |
| --- |
| void os\_run\_work(struct os\_workqueue \*wq) |

The function will not return until the work queue is flushed via a call to os\_flush\_workqueue().

## os\_flush\_workqueue()

Called to flush a work queue.

|  |
| --- |
| void os\_flush\_workqueue(struct os\_workqueue \*wq) |

Flushing a work queue stops any new work from being queued and signals the os\_run\_work() function to exit when the previously queued work has been completed.

## os\_queue\_work()

Used to queue work on a specified work queue. Pointers to both are passed as input.

|  |
| --- |
| int os\_queue\_work(struct os\_workqueue \*wq, struct os\_work \*work) |

Returns true if work is added to the work queue, false if the work is not (for example, if work is already queued or work queue is destroyed). The work struct must have been initialized before passing it to this function.

The work will be queued on the specified work queue and handled when the associated worker thread is given time to run by the scheduler.

## os\_cancel\_work()

Removes work from the specified work queue.

|  |
| --- |
| int os\_cancel\_work(struct os\_work \*work) |

Work that has been queued on a work queue may be removed while it is waiting to be carried out. This function will remove such work if it has not yet reached the head of the service queue.

Returns true if the work was pending on the work queue, false otherwise.

## os\_queue\_system\_work()

Queues work on the system work queue.

|  |
| --- |
| int os\_queue\_system\_work(struct os\_work \*work) |

The system work queue can be used by applications that have simple work functions that do not require a dedicated thread context. The work functions that are scheduled on the system work queue should be relatively simple to avoid starving others out.

## os\_init\_delayed\_work()

Initializes struct os\_delayed\_work with a work function.

|  |
| --- |
| void os\_init\_delayed\_work(struct os\_delayed\_work \*dw, os\_workfn\_t func) |

## os\_queue\_delayed\_work()

Queues work that is to be run at the specified absolute time passed as parameter expire in the future on the specified work queue.

|  |
| --- |
| int os\_queue\_delayed\_work (struct os\_delayed\_work \*dw,  struct os\_workqueue \*wq, uint32\_t expire) |

The delayed work struct must have been initialized before passing it to this function. The work will be queued on the specified work queue when the system time passes the expiration value. Returns true if the delayed work was previously pending, false otherwise.

## os\_cancel\_delayed\_work()

Cancels a pending delayed work from expiring.

|  |
| --- |
| int os\_cancel\_delayed\_work(struct os\_delayed\_work \*dw) |

Delayed work that has been scheduled to run on a work queue may be removed while waiting for its timer to expire. Returns true if the delayed work was still pending to be put on the work queue, false otherwise.

# Code Walkthrough

## Sample Application 1 – Work Queue using System Thread

### Overview

In the sample code in the path /examples/innoos\_work\_q/src/workq1.c, a work is initialized and added to the system work queue. The work is to convert the received data structure into a specific json format and print it to console output.

### Sample Code Walkthrough

Define a data structure for the data which is to be handled in the work:

|  |
| --- |
| /\*define the data for the work\*/  typedef struct my\_wq\_data\_tag  {  struct os\_work work; /\*first member should be of type struct os\_work\*/  char deviceid[64];  int current\_reading;  struct os\_thread \*generated\_by;  }my\_wq\_data; |

As shown in the code, apart from struct os\_work, this structure has a character array to hold the device ID, an integer current reading, and an os\_thread structure pointer to keep the reference of the thread which will be generating and populating this structure.

Worker function is defined as shown in the following code snippet. This is the thread function for the work and forms a simple payload (json) from the input and prints it to the console.

The work function takes a pointer to a work object, struct os\_work \*work, as input. A pointer to the full data structure defined for work can be retrieved using macro container\_of() on this pointer \*work provided to the worker function.

The macro container\_of(ptr, type,member)takes three arguments:

1. A pointer
2. Type of the container
3. Name of the member the pointer refers to.

The macro will then expand to a new address pointing to the container which accommodates the respective member.

|  |
| --- |
| static void prepare\_and\_dispatch\_device\_reading\_json(struct os\_work \*work)  {  my\_wq\_data \*my\_data;  /\*extracting the app data\*/  my\_data = container\_of(work, my\_wq\_data, work);  os\_printf ("\n{\n\t\"device\_id\":\"%s\",\n\t\"reading\":%u,\n\t\"generated\_by\":%x,\n\t\"send\_by\":%x\n}",  my\_data->deviceid,  my\_data->current\_reading,(unsignedint)my\_data->generated\_by,  (unsignedint)os\_self());  os\_msleep(25);  os\_free (my\_data);  } |

Data for the work is prepared by the application main thread.

|  |
| --- |
| /\*preparing the data for the work\*/  my\_wq\_data \*data ;  data = os\_alloc(sizeof(my\_wq\_data));  snprintf (data->deviceid,sizeof(data->deviceid),"this is exe by system thread\_%u", rand());  data->current\_reading =1;  data->generated\_by = os\_self(); |

To initialize a work and to associate a function with it, we use os\_init\_work().

Here, the member work of my\_wq\_data \*data is initialized and associated with worker function prepare\_and\_dispatch\_device\_reading\_json().

|  |
| --- |
| /\*initialize the work, associating work, and work function\*/  os\_init\_work(&data->work, prepare\_and\_dispatch\_device\_reading\_json); |

To add a work to the work queue, we use os\_queue\_system\_work().

|  |
| --- |
| /\*inserting to system work q\*/  os\_queue\_system\_work(&data->work); |

Please note that the system work queue does not need to be separately created or initialized, as it is created and initialized at the time of system start-up itself.

When the system work queue is serviced by the system thread, it will call the worker function with parameter \*work, which will eventually retrieve the full data structure associated with my\_wq\_data \*my\_data, and will perform the defined work on that data from the context of the system thread.

The work defined here is printing the data and the reference to the thread which generated the data to the console in a json format, as shown in section 5.1.4.

### Running the Application

Program workq1.elf(sdk\_x.y\examples\innoos\_work\_q\bin) 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 workq1.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

workq1.elf is created when compiling this code and 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  Workqueue Demo App 1  {  "device\_id":"this is exe by system thread\_0",  "reading":1,  "generated\_by":b2de0,  "send\_by":bede8  }  all done |

## Sample Application 2 – Work Queue using Application defined Thread

### Overview

In the sample code in the path /examples/innoos\_work\_q/src/workq2.c, two threads are created to generate requests to the work queue and two threads for handling the requests.

### Sample Code Walkthrough

Define a data structure for the data which is to be handled in the work:

|  |
| --- |
| /\*define the data structure for the workqueue\*/  typedef struct my\_wq\_data\_tag  {  Struct os\_work work;  char deviceid[64];  int current\_reading;  struct os\_thread \*generated\_by;  }my\_wq\_data; |

As shown in the code, apart from struct os\_work, this structure has a character array to hold the device ID, an integer current reading, and an os\_thread structure pointer to keep the reference of the thread which will be generating and populating this structure.

In this sample, two different work functions are defined.

First work function, prepare\_and\_dispatch\_device\_reading\_json() forms the json message from the input data.

Second work function prepare\_and\_dispatch\_device\_reading\_xml()prints to the console in XML format.

Both the work functions take the pointer to a work object, struct os\_work \*work, as input. A pointer to the full data structure defined for work can be retrieved using macro container\_of() on this pointer \*work, which is provided to the worker function, as explained in the code sample in section 5.1.

Work 1:

|  |
| --- |
| static void prepare\_and\_dispatch\_device\_reading\_json(struct os\_work \*work)  {  my\_wq\_data \*my\_data;  /\*extracting the app data\*/  my\_data = container\_of(work, my\_wq\_data, work);  os\_printf ("\n{\n\t\"device\_id\":\"%s\",\n\t\"reading\":%u,\n\t\"generated\_by\":%x,\n\t\"send\_by\":%x\n}",  my\_data->deviceid,  my\_data->current\_reading,(unsignedint)my\_data->generated\_by,(unsignedint)os\_self());  os\_msleep(25);  os\_free (my\_data);} |

Work2:

|  |
| --- |
| static void prepare\_and\_dispatch\_device\_reading\_xml(struct os\_work \*work)  {  my\_wq\_data \*my\_data;  /\*extracting the app data\*/  my\_data = container\_of(work, my\_wq\_data, work);  os\_printf ("\n{\n\t\"device\_id\":\"%s\",\n\t\"reading\":%u, \  \n\t\"generated\_by\":%x,\n\t\"send\_by\":%x\n}",  my\_data->deviceid,  my\_data->current\_reading, (unsigned int)my\_data- >generated\_by,(unsigned int)os\_self());  os\_msleep(25);  os\_free (my\_data);  } |

In this example, a separate thread is created to service the work queue where a work queue is to be initialized. To achieve this we use os\_init\_workqueue().

|  |
| --- |
| /\*work q handle\*/  struct os\_workqueue hwq;  /\*initializing work q\*/  os\_init\_workqueue(&hwq); |

Two worker threads (worker1 and worker2) are created and associated with the same work queue previously initialized.

To achieve this, os\_create\_thread()is used to create these two worker threads and a function is associated with them:

|  |
| --- |
| /\*thread 1\*/  os\_create\_thread("worker1", my\_work\_thread,&hwq,1, WORKQ\_STACK\_SIZE);  /\*thread 2\*/  os\_create\_thread("worker2", my\_work\_thread,&hwq,1, WORKQ\_STACK\_SIZE); |

The function my\_work\_thread()is associated with both the worker threads.

The work queue handle is passed as an argument to this thread function while creating both the worker threads.

The application can set the required stack size and priority for the threads based on the type of work it intends to defer to these threads.

The thread function takes the pointer to the work queue handle passed as input and calls os\_run\_work()with this work queue handle. os\_run\_work()is a blocking call and it will exit on os\_flush\_workqueue().

|  |
| --- |
| static void \*my\_work\_thread(void\*p)  {  struct os\_workqueue \*wq = p;  os\_printf ("\n%x:started worker thread",(unsignedint)os\_self());  os\_run\_work(wq);  /\* the above function will return when the workqueue is destroyed\*/  os\_printf ("\n%x:exitng worker thread",(unsignedint)os\_self());  returnNULL;} |

Two more threads (producer1 and producer2) are created in this example and are associated with the same work queue initialized previously. These producer threads will produce the work for the worker threads previously created.

To achieve this, os\_create\_thread()is used to create these two producer threads and a function is associated with them:

|  |
| --- |
| int main()  {  ...  ...  producer\_thread\_1 = os\_create\_thread("producer1", workq\_load\_producer, &hwq, 1, WORKQ\_STACK\_SIZE);  producer\_thread\_2 = os\_create\_thread("producer2", workq\_load\_producer, &hwq, 1, WORKQ\_STACK\_SIZE);  os\_join\_thread(producer\_thread\_1);  os\_join\_thread(producer\_thread\_2);  ...  ...  } |

The function workq\_load\_producer()is associated with both of these producer threads. The work queue handle is passed as an argument to this thread function while creating both the producer threads.

The data for the work is prepared by these producer threads inside the thread function workq\_load\_producer(), and os\_init\_work()is used to associate the work with the two work functions in a way that alternate data goes to xml and json works.

The work is then added to the work queue using os\_queue\_work()which returns true if succeeds in adding or false if the work is already queued.

|  |
| --- |
| static void \*workq\_load\_producer(void\*p)  {  struct os\_workqueue \*wq = p;  int msg\_counter = 0;  ...  ...  while ( (msg\_counter <= MAX\_MESSAGES\_PER\_THREAD))  {  my\_wq\_data \*data ;  data = os\_alloc(sizeof(my\_wq\_data));  snprintf (data->deviceid, sizeof(data->deviceid), "abcd\_xyz\_%u", rand());  data->current\_reading = msg\_counter;  data->generated\_by = os\_self();  ...  ...  ...  if(0==(msg\_counter%2))  {  os\_init\_work(&data->work, prepare\_and\_dispatch\_device\_reading\_json);  }  else  {  os\_init\_work(&data->work, prepare\_and\_dispatch\_device\_reading\_xml);  }  /\*inserting to the workq\*/  ret = os\_queue\_work(wq, &data->work);  if (!ret)  {  os\_printf("\nrequest is in q. ret:%d", ret);  }  os\_msleep(10);  ++msg\_counter;  }  ...  ...  ...  } |

Each of the two producer threads adds work1 and work2 alternatively to the work queue until a while loop counter finishes.

os\_flush\_workqueue() is used to send a signal to the os\_run\_work()function to exit when all the previously queued work has been completed.

|  |
| --- |
| /\*for worker thread 1\*/  os\_flush\_workqueue(&hwq);  /\*for worker thread 2\*/  os\_flush\_workqueue(&hwq); |

### Running the Application

Program workq2.elf(sdk\_x.y\examples\innoos\_work\_q\bin) 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 workq2.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

workq2.elf is created when compiling this code and gives the following console output when programmed to Talaria TWO.

Note that when the two worker threads are started, send by field in each message refers to either of these worker threads, whichever serviced the message. Based on scheduling, the messages serviced by the worker threads are random.

Each reading is generated twice, once per producer thread, and is shown in generated by the field. All the even readings by both the producers are dispatched as json and all the odd ones as xml.

This is provided as an output to the console till reading 50 is generated by both the producers. Post this, the work queue is flushed, and the worker threads exit as shown in the following snippet.

|  |
| --- |
| 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  Workqueue Demo App 2  bf968:started worker thread  b35e8:started worker thread  b3ae8:started worker thread  {  "device\_id":"abcd\_xyz\_0",  "reading":0,  "generated\_by":b3fe8,  "send\_by":bf968  }  <payload>  <device\_id>  abcd\_xyz\_1774527498  </device\_id>  <reading>  1  </reading>  <generated\_by>  b3fe8  </generated\_by>  <send\_by>  b3ae8  </send\_by>  </payload>  {  "device\_id":"abcd\_xyz\_2099690603",  "reading":2,  "generated\_by":b3fe8,  "send\_by":b35e8  }  {  "device\_id":"abcd\_xyz\_1452469787",  "reading":0,  "generated\_by":b44e8,  "send\_by":bf968  }  .  .  .  .  .  .  <payload>  <device\_id>  abcd\_xyz\_790790538  </device\_id>  <reading>  49  </reading>  <generated\_by>  b3fe8  </generated\_by>  <send\_by>  b3ae8  </send\_by>  </payload>  {  "device\_id":"abcd\_xyz\_2035955618",  "reading":40,  "generated\_by":b44e8,  "send\_by":bf968  }  <payload>  <device\_id>  abcd\_xyz\_657584690  </device\_id>  <reading>  49  </reading>  <generated\_by>  b44e8  </generated\_by>  <send\_by>  b3ae8  </send\_by>  </payload>  b35e8:exiting worker thread  b3ae8:exiting worker thread  all done |