Message Queues

# Introduction

Message queues in InnoOS™ provide a simple abstraction by which threads pass messages between each other. This document briefly explains the use of message queue APIs in InnoOS™ using a sample application.

# Message Queues

A message queue is a linked list of messages where messages can be queued and dequeued by different processes/tasks, Thus providing the mechanism for inter-process/task communication.

Messages are objects that can be passed between threads using the message interface functions.

The messages are buffers allocated using the normal memory allocation function os\_alloc(), with the addition of a specific header at the beginning of the message that is defined by the os\_msg struct.

Each message is given an identification or type so that processes can select the appropriate message.

Message Queue is the low-level abstraction upon which thread message queue APIs os\_sendmsg() and os\_recvmsg() are based.

# Message Queue APIs

## os\_sendmsg ()

Post a message to a thread using the current thread as a sender

|  |
| --- |
| void os\_sendmsg(struct os\_thread \*thread, struct os\_msg \*msg) |

## os\_recvmsg()

Retrieves the next message from the threads message queue, optionally blocking the thread until a message is available.

|  |
| --- |
| struct os\_msg \* os\_recvmsg(bool noblock) |

Returns a pointer to os\_msg.

## os\_msg\_alloc()

Allocates a message (including the message header).

|  |
| --- |
| struct os\_msg\* os\_msg\_alloc(unsigned int type, size\_t size) |

Parameters passed include:

1. Arbitrary integer that identifies the message type
2. Size of the message (including the size for the message header)
3. Pointer to newly allocated message or NULL.

## Macro os\_msg\_alloc()

Provides a convenient way to allocate a message of a specific type and returns a pointer with that type. Parameters used are:

1. \_msg: Pointer variable to which the allocated message will be assigned
2. \_type: Message-ID as an integer
3. \_extra: Amount of additional buffer space to be allocated (in addition to the size of the message type)

## os\_msg\_release()

Frees a message previously allocated using os\_msg\_alloc().

|  |
| --- |
| os\_msg\_release(struct os\_msg \*msg) |

# Code Walkthrough

## Overview

This sample code /examples/innoos\_msg\_q/src/msg\_q.c demonstrates the use of message queues in InnoOS™.

The sample code contains two threads:

* Message generator (sensor thread)

This thread simulates reading sensor values. It generates data including sensor reading and sensor ID. This data is added to the message queue for further processing. Sensor values are generated as random numbers between 0 and 200.

* Message processor (consumer thread)

This thread picks messages from the queue and prints message data on the console. It also frees message memory after processing a message.

In this example, the sensor thread acts as a producer of data which is consumed by the message processor. A message queue facilitates the passing of data between these two threads. There can be more than one producer and consumer using a message queue.

## Sample Code Walkthrough

### Message Structure

As described previously, message queues are useful for exchanging data between threads. Data is defined by the application. In this example, data consists of a sensor ID and a sensor reading:

|  |
| --- |
| typedef struct message\_data{  struct os\_msg msg; /\*This needs to be the first member of the structure\*/  char sensor\_id[MSG\_QUE\_MAX\_SENSOR\_ID\_LEN];  int sensor\_reading;  }  messageData; |

When defining the message data structure, struct os\_msgmust be the first member. The application can add additional members after struct os\_msg as required.

### Creating Sensor and Consumer Threads

The sensor and consumer threads are created as follows:

|  |
| --- |
| /\*Createconsumerthread\*/  consumer\_thread = os\_create\_thread("consumer\_thread",consumer\_thread\_func,  NULL, MSG\_QUE\_THREAD\_PRIORITY, MSG\_QUE\_THREAD\_STACK\_SIZE);  if( NULL == consumer\_thread ){  os\_printf("\nCreation of consumer\_thread failed!. Error:%s", strerror(errno));  return -1;  }  /\*Create sensor thread\*/  sensor\_thread = os\_create\_thread("sensor\_thread",sensor\_thread\_func,  NULL, MSG\_QUE\_THREAD\_PRIORITY, MSG\_QUE\_THREAD\_STACK\_SIZE);  if( NULL == sensor\_thread ){  os\_printf("\nCreation of sensor\_thread failed!. Error:%s", strerror(errno));  return -1;  } |

### Sensor Thread Procedure

The sensor thread simulates reading from a sensor using a random value as the sensor reading. The thread allocates a message using the os\_msg\_alloc()API and populates the message with sensor data. The message is pushed onto the message queue using os\_sendmsg().

|  |
| --- |
| /\* Sensor thread callback function. This thread creates messages with random sensor readings. \*/  static void\* sensor\_thread\_func(void\* arg)  {  …  messageData \*msg\_info;  msg\_info = (messageData \*) os\_msg\_alloc(MSG\_QUE\_MSG\_TYPE\_SENSOR\_DATA,  sizeof(\*msg\_info));  /\* Set dummy data \*/  snprintf(sensor\_buf,18,"ABCDEFGHIJKLMN-%d",nMsgSent);  strncpy(msg\_info->sensor\_id, sensor\_buf, sizeof(msg\_info->sensor\_id));  /\*Setting a random value\*/  msg\_info->sensor\_reading = (int) (rand()%MSG\_QUE\_MAX\_SENSOR\_POSSIBLE\_VALUE);  /\* Send the message over Queue\*/  os\_sendmsg(consumer\_thread, &msg\_info->msg);  os\_printf ("\n%x:Number of messages sent %d",\  MSG\_QUE\_CURRENT\_THREAD\_ID, ++nMsgSent); |

### Consumer Thread Procedure

The consumer thread checks for messages in the queue. The os\_recvmsg() function receives the messages from the message queue optionally blocking the thread based on the argument passed. This function returns a struct os\_msg pointer.

A pointer to the application-defined data structure can be retrieved using container\_of() API.

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.

The consumer thread is then able to perform any processing of the message data. In this case, the data is printed to the console. After the message has been processed, it is freed with os\_msg\_release().

|  |
| --- |
| /\* Consumer thread callback function \*/  static void\* consumer\_thread\_func(void\* arg)  …  struct os\_msg \*m = os\_recvmsg(false);  …  /\* Extract the application-specific message info from the message \*/  msg\_info = container\_of(m, /\* Received message pointer \*/  messageData /\* Expected message data type \*/,  msg); /\* Variable name for 'struct os\_msg' in 'messageData' \*/  …  os\_printf("\n%x:Got new msg. Count[%d]. Reading[%d] ID[%s]",  MSG\_QUE\_CURRENT\_THREAD\_ID, nMsgReceived,  msg\_info->sensor\_reading, msg\_info->sensor\_id);  …  /\* Release the memory\*/  os\_msg\_release(m);  } |

## Running the Application

Program msg\_q.elf (sdk\_x.y\examples\innoos\_msg\_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 msg\_q.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

msg\_q.elf is created when compiling the 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  Innos Msg Queue Demo App  b4190:Consumer thread started  b4410:Number of messages sent 1  b4190:Got new msg. Count[1]. Reading[0] ID[ABCDEFGHIJKLMN-0]  b4410:Number of messages sent 2  b4190:Got new msg. Count[2]. Reading[187] ID[ABCDEFGHIJKLMN-1]  ….  ….  ….  b4410:Number of messages sent 19  b4190:Got new msg. Count[19]. Reading[190] ID[ABCDEFGHIJKLMN-18]  b4410:Number of messages sent 20  b4190:Got new msg. Count[20]. Reading[131] ID[ABCDEFGHIJKLMN-19]  b4190:Received all messages from the producer  b4410:Completed sending all the messages. |