Exercise 18 Zero Copy Mailbox

This exercise demonstrates the configuration of a memory pool and message queue to transfer complex data between threads.

In the Pack Installer select “Ex 18 Memory Pool” and copy it to your tutorial directory.

This exercise creates a memory pool and a message queue. A producer thread acquires a buffer from the memory pool and fills it with data. A pointer to the memory pool buffer is then placed in the message queue. A second thread reads the pointer from the message queue and then accesses the data stored in the memory pool buffer before freeing the buffer back to the memory pool. This allows large amounts of data to be moved from one thread to another in a safe synchronized way. This is called a ‘zero copy’ memory queue as only the pointer is moved through the message queue, the actual data does not move memory locations.

At the beginning of main.c the memory pool and message queue are defined.

```
static const osMemoryPoolAttr_t memorypoolAttr_mpool ={

        .name = "memory_pool",

};

void app_main (void *argument) {

  mpool = osMemoryPoolNew(16, sizeof(message_t),&memorypoolAttr_mpool );

  queue = osMessageQueueNew(16,4, NULL);

  osThreadNew(producer_thread, NULL, &ThreadAttr_producer);

  osThreadNew(consumer_thread, NULL, &ThreadAttr_consumer);

}
```

In the producer thread acquire a message buffer, fill it with data and post a testData++;

```
while (1){

        if(testData == 0xAA){

                testData = 0x55;

        }

        else{

                testData = 0xAA;

        }

        message = (message_t*)osMemoryPoolAlloc(mpool,osWaitForever); //Allocate a memory pool buffer

        for(index =0;index<8;index++){
```

```
            message->canData[index] = testData;
        }

        osMessageQueuePut(queue, &message,NULL, osWaitForever);
        osDelay(1000);

    }
```

Then in the consumer thread we can read the message queue to get the next pointer and then access the memory pool buffer. Once we have used the data in the buffer it can be released back to the memory pool.

```
        while (1) {

            osMessageQueueGet(queue,&message,NULL,osWaitForever);

            LED_SetOut((uint32_t)message->canData[0]);
            osMemoryPoolFree(mpool, message);

        }
```

Build the code and start the debugger.

Place breakpoints on the osMessagePut and osmessageGet functions.

```
41          osMessageQueuePut(queue, &message,NULL, osWaitForever);
42          osDelay(1000);
43      }

56      osMessageQueueGet(queue,&message,NULL,osWaitForever);
57      LED_SetOut((uint32_t)message->canData[0]);
```

Fig 54 Set breakpoints on the sending and receiving threads

Run the code and observe the data being transferred between the threads.