# **Embedded Systems Essentials with Arm: Get Practical with Hardware**

## Module 4

## KV1: Further Features of the Mbed RTOS

In the previous module, you learned that we allow the OS to decide how resources are allocated to a specific task. If we have a shared resource, we might get into a tricky situation where one thread might be using a specific resource, which another thread might need to use. Things can go wrong if a resource is left in an incomplete condition. That resource could be a block of software or it could be a piece of hardware such as a printer or a peripheral.

To counter this issue, we introduce the concept of a mutex (for mutual exclusion).

The mutex allows access to a shared resource between two threads. This also allows for synchronization to take place because one thread can stop or start another thread if it blocks access to a needed resource.

The concept of a deadlock is important to take note of in RTOSs. This happens when several threads block off each other because they’re waiting for each other. For the purposes of this course, it’s important to know that the concept exists but we won’t delve into it just yet. As your skills develop, you can look into that.

The Mbed RTOS has a number of APIs that we can use to create mutexes. This is a subset of what’s available to us.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| **Mutex** (const char \*name)​ | Create and Initialize a Mutex object. ​ |
| void **lock** ()​ | Wait until a Mutex becomes available. ​ |
| bool **trylock** ()​ | Try to lock the mutex, and return immediately..​ |
| void **unlock** ()​ | Unlock the mutex that has previously been locked by the same thread​ |

As always, check the website for the most current list of APIs.

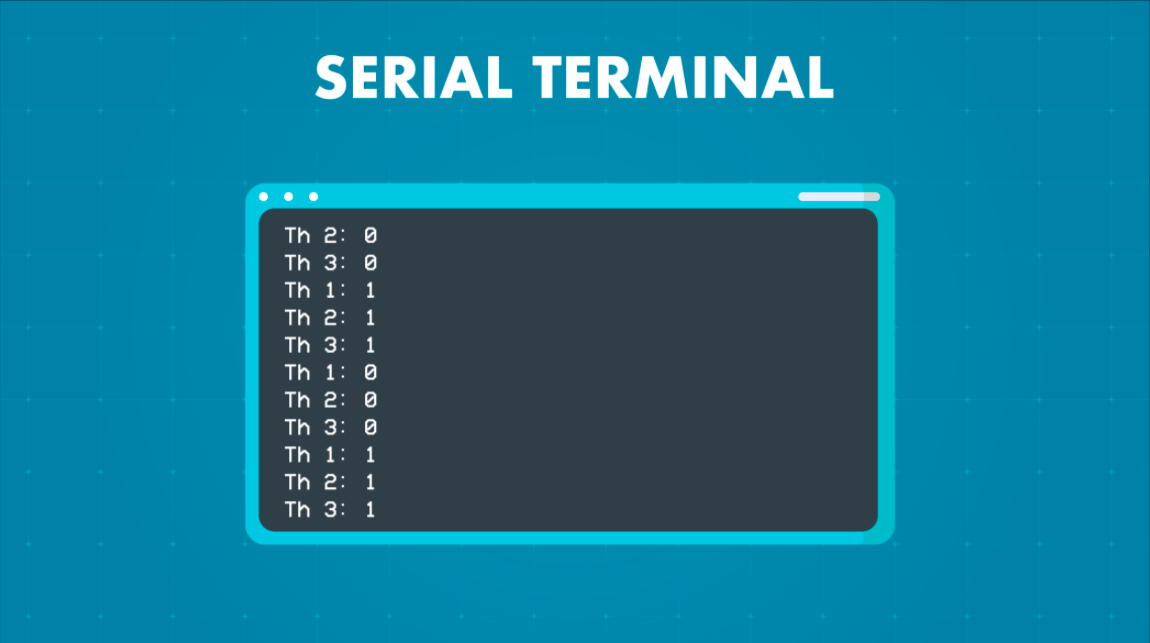
Let’s now look at a simple example of how we can create a mutex.

In this example, we have three threads and a single mutex:

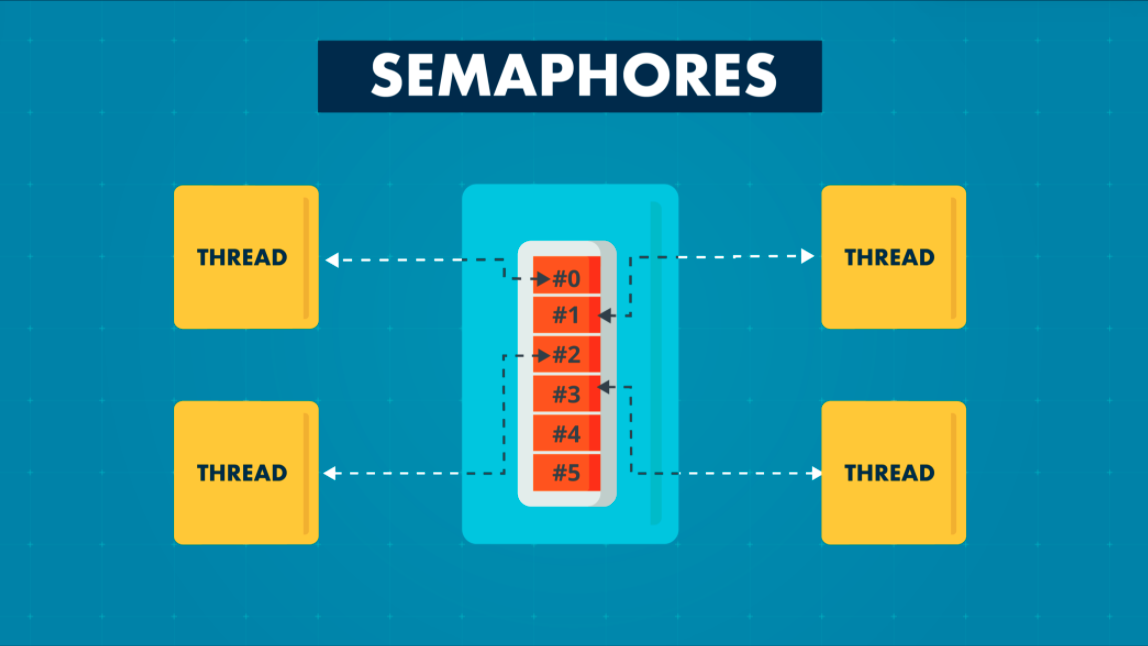
|  |
| --- |
| #include "mbed.h"​  ​  Mutex stdio\_mutex; //Create Mutex named stdio\_mutex​  Thread t2;​  Thread t3;​  ​  void notify(const char \*name, int state) {​  stdio\_mutex.lock();​  printf("%s: %d\n\r", name, state);​  stdio\_mutex.unlock();​  }​  void test\_thread(void const \*args) {​  while (true) {​  notify((const char \*)args, 0);​  ThisThread::sleep\_for(1000);​  notify((const char \*)args, 1);​  ThisThread::sleep\_for(1000);​  }​  }​  int main(){​  t2.start(callback(test\_thread, (void \*)"Th 2"));​  t3.start(callback(test\_thread, (void \*)"Th 3"));​  test\_thread((void \*)"Th 1");​  } |

Remember that your first thread is always Main. Here, the second and third threads are labeled t2 and t3. At the beginning here, we’ve created a mutex which we’ve called stdio\_mutex.

When we look at Main, we see that t2 and t3 use the callback technique to call test\_thread. Notice that all three threads invoke test\_thread. The test\_thread itself calls notify. It locks the mutex and then invokes a printer, and then unlocks the mutex.

When the program is run on an Mbed device and connected to a serial terminal, it gives this outcome. 

Let’s move onto the concept of semaphores. A semaphore extends the mutex concept, but manages and protects access to several shared resources. It’s a variable of abstract data type. Unlike a mutex, it doesn’t have the concept of an owner. For example, a semaphore enables access to, and management of, a group of identical peripherals.

In this diagram, we see multiple threads wanting access to a shared resource. 

This is a simplification. In the real world there may be multiple threads and multiple resources. The semaphore manages access to shared resources, and, as a by-product of that, also manages the ability to synchronize activity.

This is a subset of example functions from the Mbed API for a semaphore.

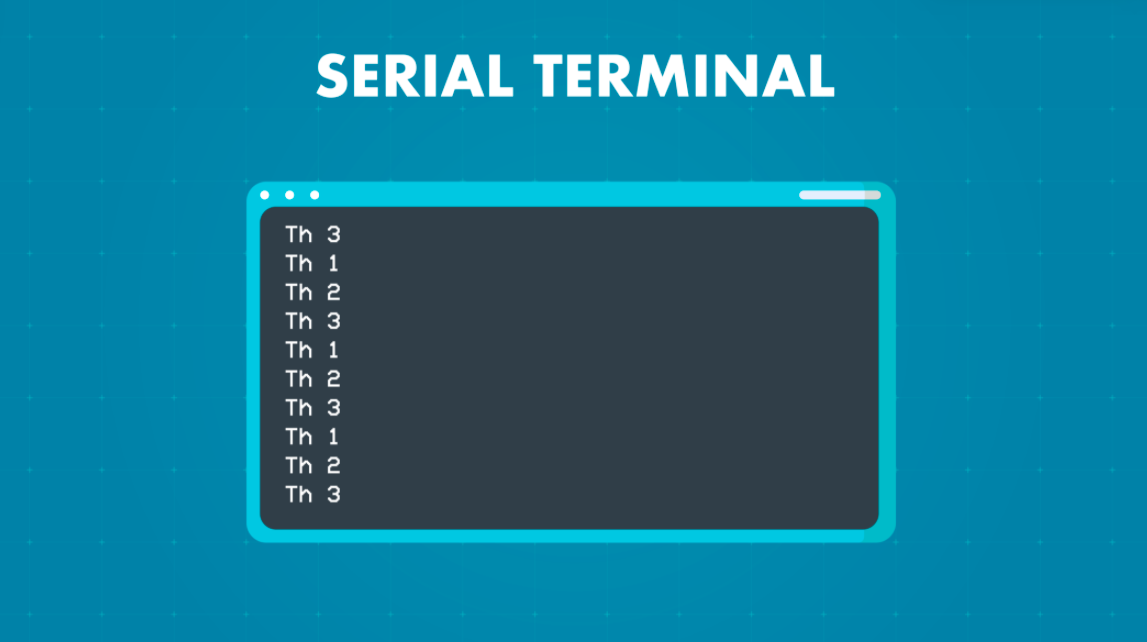
|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| **Semaphore** (int32\_t count=0)​ | Create and Initialize a **Semaphore** object used for managing resources, where count is number of available resources.​ |
| void **acquire** ()​ | Wait until a **Semaphore** resource becomes available.​ |
| bool **try\_acquire** ()​ | Try to acquire a **Semaphore** resource, and return immediately. Returns true if a resource was acquired, false otherwise.​ |
| osStatus **release** (void)​ | Release a **Semaphore** resource that was obtained with acquire().​ |

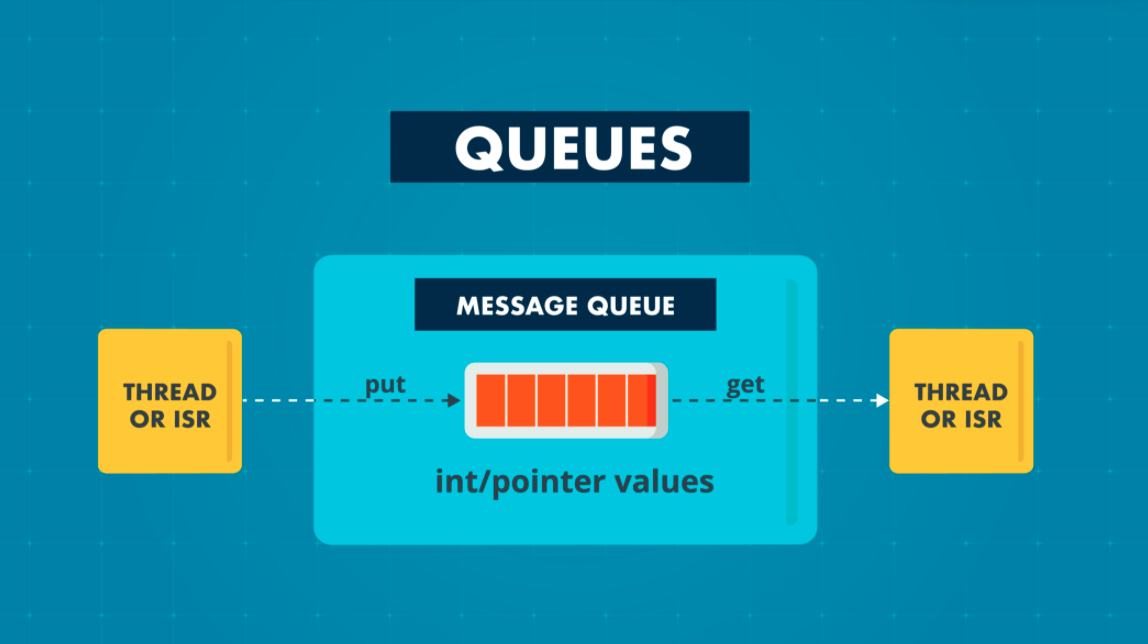
As you develop your skills, you may want to use these first.

This simple example shows how we can use a semaphore to manage thread access to the printf() function.

|  |
| --- |
| #include "mbed.h"​  ​  Semaphore one\_slot(1); //Create semaphore, named one\_slot, one resource​  Thread t2;​  Thread t3;​  ​  void test\_thread(void const \*name) {​  while (true) {​  one\_slot.acquire(); //acquire semaphore​  printf("%s\n\r", (const char \*)name);​  ThisThread::sleep\_for(1000);​  one\_slot.release(); //release semaphore​  }​  }​  ​  int main(void) {​  t2.start(callback(test\_thread, (void \*)"Th 2"));​  t3.start(callback(test\_thread, (void \*)"Th 3"));​  test\_thread((void \*)"Th 1");​  } |

As only a single resource is shared, the resulting behavior is similar to the mutex action in the previous example.

Again, this is a three-thread example, with Main as our first thread. Then we have t2 and t3. All three threads invoke the printf() function and will each print their names on the display.

Let’s have a look at passing data between threads and the concept of queues. This diagram shows how one thread or an ISR can put data into a queue and then another thread can extract that data. 

This is much like a data pipeline.

This is a subset of what’s available to use in the Mbed RTOS.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| **Queue** ()​ | Create and initialize a message Queue of objects.​ |
| bool **empty** () const​ | Check if the queue is empty.​ |
| bool **full** () const​ | Check if the queue is full.​ |
| osEvent **get** (uint32\_t millisec=osWaitForever)​ | Get a message or wait for a message from the queue. The message is stored in the value field of the returned osEvent object.​ |
| osStatus **put** ()​ | Inserts the given element to the end of the queue.​ |

We can also combine the concept of queues with that of a memory pool, which allows you to manage memory and blocks of data that you want to keep hold of.

These are a few example functions from the Mbed API for memory pools.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| MemoryPool()​ | Create and Initialize a memory pool, of data type T and number of objects queue\_sz​ |
| T\* **alloc**()​ | Allocate a memory block from a memory pool, without blocking.​ |
| osStatus free()​ | Free a memory block.​ |

Here’s an example that’s a bit more complex than the last two we saw.

|  |
| --- |
| /\* Copyright (c) 2006-2020 Arm Limited and affiliates.​  \* SPDX-License-Identifier: Apache-2.0 \*/​  #include "mbed.h"​  typedef struct {​  float voltage; /\* AD result of measured voltage \*/​  float current; /\* AD result of measured current \*/​  uint32\_t counter; /\* A counter value \*/​  } message\_t; //message\_t is structure with three variables​  ​  MemoryPool<message\_t, 16> mpool;​  Queue<message\_t, 16> queue;​  Thread thread;  void send\_thread(void) { // Send Thread​  uint32\_t i = 0;​  while (true) {​  i++; // simulated data update, eg reading an ADC​  message\_t \*message = mpool.alloc();​  message->voltage = (i \* 0.1) \* 33;​  message->current = (i \* 0.1) \* 11;​  message->counter = i;​  queue.put(message); //put message at end of queue​  ThisThread::sleep\_for(1000);​  }​  }​  int main(void){​  thread.start(callback(send\_thread));​  while (true) {​  osEvent evt = queue.get();​  if (evt.status == osEventMessage) {​  message\_t \*message = (message\_t \*)evt.value.p;​  printf("\nVoltage: %.2f V\n\r", message->voltage);​  printf("Current: %.2f A\n\r", message->current);​  printf("Number of cycles: %u\n\r", message->counter);​  mpool.free(message);​  }​  }​  } |

We’re generating data, storing that in a memory pool, then we’re transferring that data through the queue.

At the beginning of the program we have the variable i. That’s our simulated data. In this program, we’re creating pseudo data and putting that into a queue, which we’ve created here. The name of the queue is queue. We have just one thread alongside.

This is the example output of the program viewed on a serial terminal.

**Visit arm.com for more information.**