

# Lab instructions – Coursework 2

## Threading and Communication

### Communication with host

So far, you have sent messages to the host using the serial port with `printf()` commands in the main function. You also need the ability to receive messages and you may want to send debugging messages from other tasks in the code. It makes sense to consider the communication as distinct tasks to avoid problems with sharing the serial port resource between different tasks. Placing the tasks in separate threads makes it easy to handle waiting for communication events and also makes it easier to analyse the latency of the task.

### Outgoing communication

Our first thread will receive messages from other parts of the code and write them to the serial port.

1. Import the RTOS library into your code:  
[https://os.mbed.com/users/mbed\\_official/code/mbed-rtos/](https://os.mbed.com/users/mbed_official/code/mbed-rtos/) Don't search for the library within the compiler because it returns an out-of-date version. Use 'Import into Compiler' on the library webpage at the link.
2. In the global variables section of your code, create an instance of the class Thread (<https://os.mbed.com/handbook/RTOS>) called `commOutT`. Also create a function with the prototype `void commOutFn()`. Start the thread within `main()` by calling `commOutT.start(commOutFn)`. Move the instantiation of Serial object `pc` to the global part of the file.
3. Create a global instance `outMessages` of the class `Mail`. `Mail` is a data structure that allows you to pass information between threads. It combines two functions: first, it provides a FIFO queue that can be written to or read from in a thread-safe manner. Second, it maintains a pool of memory units to contain the information being passed and avoiding the need for frequent dynamic memory operations. In this case, it will allow concurrent threads to independently send data back to the host.

`Mail` is a C++ template, which means it is customised on instantiation to use a data type of your choosing. We shall create our own data type `message_t` consisting of code and data variables. We'll allow up to 16 messages to be queued up, which is the second template parameter of `Mail`.

```
typedef struct{
    uint8_t code;
    uint32_t data;
} message_t ;

Mail<message_t,16> outMessages;
```

4. It's worth creating a separate function to add messages to the queue. First, you need to call `Mail::alloc()` to return a pointer to the memory that will be used to store the message. Then, the code and data need to be written into that data structure and, finally, `Mail::put()` places the message pointer in the queue. The function appears below; use it to replace the `printf()` calls in `main()`. Allocate a unique code integer to each message type (e.g. report a bitcoin nonce); you could use a custom enum type to do this allocation

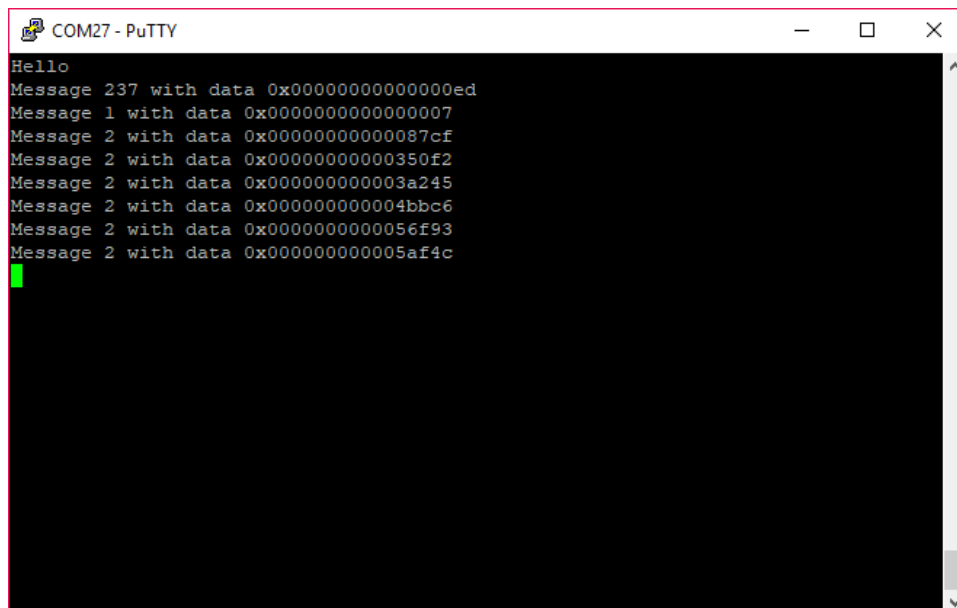
automatically if you like. Note that you are not queuing a message string, just a code and data value that will be formatted into a string later.

```
void putMessage(uint8_t code, uint32_t data){
    message_t *pMessage = outMessages.alloc();
    pMessage->code = code;
    pMessage->data = data;
    outMessages.put(pMessage);
}
```

5. The function of the thread function commOutFn() is to take messages from the queue and print them on the serial port. So it should be an infinite loop that repeatedly calls Mail::get(), which waits for a message to be available in the queue. Mail::get() returns type osEvent, a data structure that contains a pointer to the message itself. The pointer is dereferenced to access the values of code and data, which are then printed to the serial port. Finally, the memory unit that was used is returned to the pool using Mail::free(). You could test the value of code if you want to generate a customised message for different events.

```
while(1) {
    osEvent newEvent = outMessages.get();
    message_t *pMessage = (message_t*)newEvent.value.p;
    pc.printf("Message %d with data 0x%016x\n",
        pMessage->code, pMessage->data);
    outMessages.free(pMessage);
}
```

6. Test it! The functionality should be similar to before but now you can send messages from different threads as you develop your code. Your putMessage() function is reentrant because the methods of Mail are documented as thread-safe and all other variables are local. The example below shows three message types – Message 2 is reporting nonce matches.

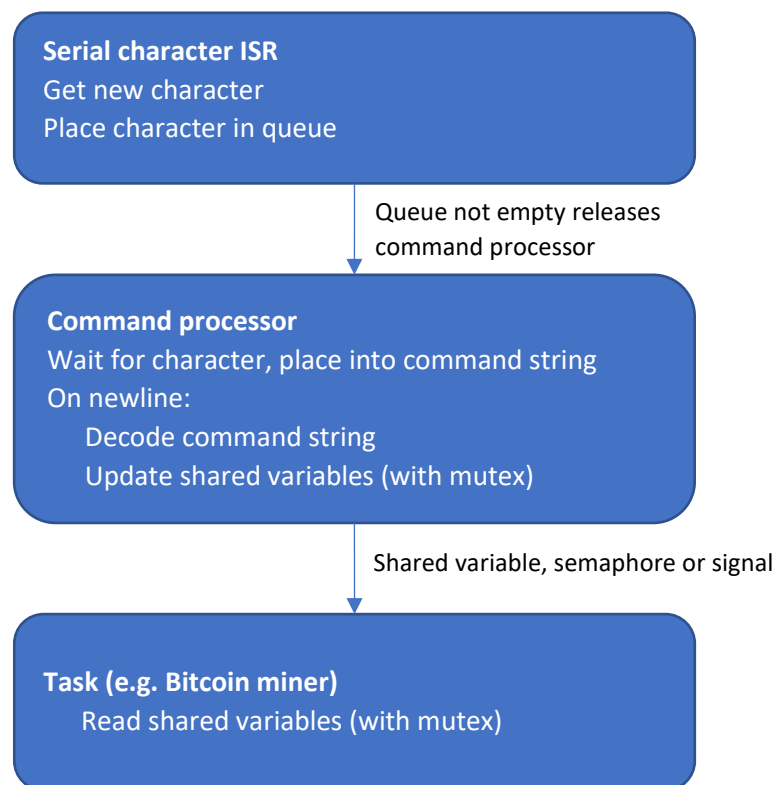


```
COM27 - PuTTY
Hello
Message 237 with data 0x00000000000000ed
Message 1 with data 0x0000000000000007
Message 2 with data 0x000000000000087cf
Message 2 with data 0x000000000000350f2
Message 2 with data 0x00000000000003a245
Message 2 with data 0x0000000000004bbc6
Message 2 with data 0x00000000000056f93
Message 2 with data 0x0000000000005af4c
```

## Incoming communication

The specification requires you to act on certain commands sent by the serial port. One of these is to set the Bitcoin mining key to a specified value and we'll begin by decoding this command. Bytes of serial data arrive asynchronously so they are best handled by an interrupt. The ISR should be as short as possible so the bytes will be placed into a queue for later decoding by a dedicated thread.

The following diagram shows the control dependency between the three different tasks and the relevant behaviour in each task.



1. Change the class of pc from `Serial` to `RawSerial`; this version is better for us because `Serial` contains some undocumented buffering behaviour.
2. Create an ISR that will receive each incoming byte and place it into a queue.
  - a. Method `uint8_t RawSerial::getc()` retrieves a byte from the serial port. This must be called to prevent the interrupt from retriggering.
  - b. Create a global instance of class `Queue<void, 8>` to buffer incoming characters. `Queue` is normally used to pass pointers to data structures and in the case the pointers will be type `void*`, since `Queue` has been templated with `void`. We are only passing single bytes in this case so we can just cast the character to a pointer and put it on the queue directly:

```
void serialISR(){
    uint8_t newChar = pc.getc();
    inCharQ.put((void*)newChar);
}
```

3. Create a new thread to decode commands. In the thread function, begin by attaching the new ISR to serial port events, then use method `Queue::get()` to wait for each new character:

```
pc.attach(&serialISR);
while(1) {
    osEvent newEvent = inCharQ.get();
    uint8_t newChar = (uint8_t)newEvent.value.p;
    ...
}
```

Handle each new character as follows:

- a. Place it on the end of a `char[]` array. Make the array large enough to contain the longest possible command. You will need to keep an index of the current buffer position. This would be easier with the C++ `std::string` class but there is not enough memory available for this. Using a C-style array is also faster and does not involve the uncertainty of dynamic memory allocation.
- b. Include a test to make sure you do not write past the end of the buffer if the incoming string is too long. Buffer overflows have historically been the source of many software crashes and vulnerabilities since you may be able to alter the return address of a function if you write past the end of a buffer.
- c. If the incoming character is a carriage return `'\r'` it indicates the end of a command. At this point:
  - i. Place a string termination character `'\0'` at the end of the command. This is used by functions like `printf()` and `scanf()` to detect the end of the string.
  - ii. Reset the buffer position index back to 0 ready to record the next command.
  - iii. Test the first character to determine which command was sent.
  - iv. Decode the rest of the command
4. Begin by decoding the command to set the bitcoin key, which is the letter K followed by 16 hexadecimal digits.
  - a. Create a new global variable `volatile uint64_t newKey`, which will be used to pass the key from the command decoder to the bitcoin miner. Also create a Mutex `newKey_mutex`, which will prevent simultaneous access of `newKey` by the command decoder and the bitcoin miner.
  - b. Add statements to decode the command and update `newKey` once a carriage return is detected. The mutex is obtained while the variable is being accessed:

```
if newCmd[0] == 'K'{
    newKey_mutex.lock();
    sscanf(newCmd, "K%x", &newKey); //Decode the command
    newKey_mutex.unlock();
}
```

- c. Make changes to the bitcoin miner to use the new key. Before every hash attempt, copy `newKey` into `key`. Remember that `key` is a pointer to data within `sequence[]` so it must be dereferenced. The reading of `newKey` should be protected with the mutex in the same way as the writing.
5. Add a statement in `main` to start the new command decoder thread. Test by adding calls to `putMessage()` to confirm the command has been decoded. The set of successful nonces that you find will depend on the value of `key`.