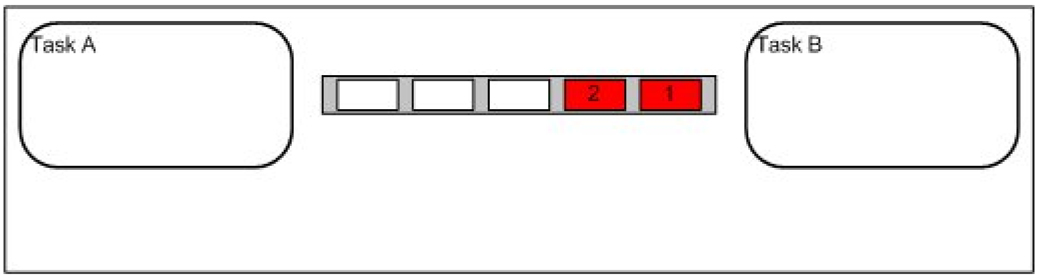
FreeRTOS Queues

[Inter-task communication and synchronisation]

FreeRTOS Queues

[See also Blocking on Multiple RTOS Objects]

Queues are the primary form of intertask communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.



Writing to and reading from a queue. In this example the queue was created to hold 5 items, and the queue never becomes full.

User Model: Maximum Simplicity, Maximum Flexibility . . .

The FreeRTOS queue usage model manages to combine simplicity with flexibility - attributes that are normally mutually exclusive. Messages are sent through queues by copy, meaning the data (which can be a pointer to larger buffers) is itself copied into the queue rather than the queue always storing just a reference to the data. This is the best approach because:

- Small messages that are already contained in C variables (integers, small structures, etc.) can be sent into a queue directly. There is no need to allocate a buffer for the message and then copy the variable into the allocated buffer. Likewise, messages can be read from queues directly into C variables.

  Further, sending to a queue in this way allows the sending task to immediately overwrite the variable or buffer that was sent to the queue, even when the sent message remains in the queue. Because the data contained in the variable was copied into the queue the variable itself is available for re-use. There is no requirement for the task that sends the message and the task that receives the message to agree which task owns the message, and which task is responsible for freeing the message when it is no longer required.

- Using queues that pass data by copy does not prevent queues from being used to pass data by reference. When the size of a message reaches a point where it is not practical to copy the entire message into the queue byte for byte, define the queue to hold pointers and copy just a pointer to the message into the queue instead. This is exactly how the FreeRTOS-Plus-UDP implementation passes large network buffers around the FreeRTOS IP stack.

- The kernel takes complete responsibility for allocating the memory used as the queue storage area.

- Variable sized messages can be sent by defining queues to hold structures that contain a member that points to the queued message, and another member that holds the size of the queued message.

- A single queue can be used to receive different message types, and messages from multiple locations, by defining the queue to hold a structure that has a member that holds the message type, and another member that holds the message data (or a pointer to the message data). How the data is interpreted depends on the message type. This is exactly how the task that manages the FreeRTOS-Plus-UDP IP stack is able to use a single queue to receive notifications of ARP timer events, packets being received from the Ethernet hardware, packets being received from the application, network down events, etc.

- The implementation is naturally suited for use in a memory protected environment. A task that is restricted to a protected memory area can pass data to a task that is restricted to a different protected memory area because invoking the RTOS by calling the queue send function will raise the microcontroller privilege level. The queue storage area is only accessed by the RTOS (with full privileges).

- A separate API is provided for use inside of an interrupt. Separating the API used from an RTOS task from that used from an interrupt service routine means the implementation of the RTOS API functions do not carry the overhead of checking their call context each time they execute. Using a separate interrupt API also means, in most cases, creating RTOS aware interrupt service routines is simpler for end users than when compared to alternative RTOS products.

- In every way, the API is simpler.

The FreeRTOS tutorial book provides additional information on queues, binary semaphores, mutexes, counting semaphores and recursive semaphores, along with simple worked examples in a set of accompanying example projects.

Blocking on Queues

Queue API functions permit a block time to be specified.

When a task attempts to read from an empty queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either data becomes available on the queue, or the block time expires.

When a task attempts to write to a full queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either space becomes available in the queue, or the block time expires.

If more than one task block on the same queue then the task with the highest priority will be the task that is unblocked first.

See the Queue Management section of the user documentation for a list of queue related API functions. Searching the files in the `FreeRTOS/Demo/Common/Minimal` directory will reveal multiple examples of their usage.

Note that interrupts must NOT use API functions that do not end in "FromISR".