EE458 - Embedded Systems
Queues/Reentrancy

• Outline
  – Introduction to Queues
  – Queue Operations
  – Queue Applications
  – RTEMS Queues
  – Reentrancy

• References
  – RTC: Chapter 7
  – CUG: Chapter 14
Queues/Reentrancy

Introduction

- A *message queue* is a buffer object through which tasks and ISRs send and receive messages. Message queues can be used for communication and for *synchronization*.
- The queue will accept and hold messages from a sender until the receiver is ready to read them. The sender does not have to wait until the receiver is ready for the message.
Queues/Reentrancy
Introduction

• You may create and use several different message queues in your application. Each queue has a unique ID associated with it.

• There is a maximum number of messages that a queue can hold. There is also a limit on the message size. It is usually more efficient to pass a pointer to a memory area containing a large message than it is to pass the message directly via the queue.
Queues/Reentrancy
Introduction

• There are two waiting task lists associated with a queue. Receiving tasks may block in the receiving task wait list if the queue is empty. Sending tasks may block in the sending task wait list if the queue is full. (In RTEMS a sending task will not block, an error is returned if the queue is full.)

• Most RTOSes optionally allow a sending or receiving task to receive an error code instead of blocking.
Queues/Reentrancy
Queue Operations

- Most RTOSes have the following queue related routines:
  - create: create a queue
  - delete: delete the queue
  - send: add a message to the queue
  - receive: read a message from the queue
  - info: get info on the number of messages waiting
Queues/Reentrancy

Queue Creation and Deletion

- Queues are treated as global objects and are not owned by a particular task.
- The size of the queue may be determined either by configuration or at runtime, depending on the RTOS.
- When a queue is deleted all messages are discarded. If the queue is empty and there are tasks on the receiving wait lists, the receive call returns with an error.
Queues/Reentrancy
Sending Messages

• Messages are typically queued in FIFO order. Some RTOSes support LIFOs and message priorities.

• When sending, different routines may be used to determine what the task will do if the queue is full: block forever, block with a timeout, not block.

• An ISR sending a message to a queue should never block.
Queues/Reentrancy
Receiving Messages

• Receiving tasks can receive messages with different receive policies: block forever, block with a timeout, not block. A receive task will block when the queue is empty.

• If a message is already in the queue when a task makes a receive call, the task returns with the message immediately.

• When a message is received it is automatically removed from the queue.
Queues/Reentrancy
Queue Applications

• Typically a queue is used for non-interlocked, one-way data communication. There is one sending task and one receiving task.

• A queue and a semaphore can be used for interlocked, one-way data communication. The sending task sends the message and then blocks on a semaphore. After receiving a message, the receive task releases the semaphore.
Queues/Reentrancy
Queue Applications

- Two queues can be used for *interlocked*, *two-way data communication* (full-duplex communication). This can be useful in client/server applications. Clients send messages to the server queue. The server responds using a client queue. (There is a unique queue for each client.)

- Some RTOSes allow *broadcast* messages to be sent to all tasks waiting on the queue.
Queues/Reentrancy
RTEMS Queues

- RTEMS queues can be created to hold an arbitrary number of messages. Messages may be of any length.
- Messages can be queued in FIFO order with the `rtems_message_queue_send()` routine. The `rtems_message_queue_urgent()` routine can be used to post in LIFO order.
- Depending on an attribute, receive tasks are dequeued in either FIFO or PRIORITY order.
Queues/Reentrancy

RTEMS Queues

- The RTEMS queue receive directive accepts the RTEMS_WAIT and RTEMS_NO_WAIT options. If the WAIT option is used, a timeout in ticks can be specified.
- There is a directive to determine the number of messages that are contained in a queue.
- Broadcast and flush routines are provided.
- See the CUG for details.
Queues/Reentrancy
RTEMS Queues

- In addition to correctly defining the macro `CONFIGURE_MAXIMUM_MESSAGE_QUEUES` you must also define the macro `CONFIGURE_MESSAGE_BUFFER_MEMORY` properly.

- This last macro must be defined as the number of bytes to be reserved for all messages in all queues. The helper macro `CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE` can be used. (See CUG configuration).
#define CONFIGURE_MESSAGE_BUFFER_MEMORY \ 
(CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE( \ 
  24, sizeof(one_message_type) + \ 
  CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE( \ 
    500, sizeof(other_message_type) \ 
  )
)
• A reentrant function is one that works correctly when called by multiple tasks.

• A function will normally be reentrant if all variables used by the function are stored on the stack (non-static local variables). Functions that use global variables or static local variables are, in general, not reentrant.

• You should try to write all functions to be reentrant if possible.
Queues/Reentrancy

Reentrancy

- Here's an example of a non-reentrant function:

```c
int temp;
void swap(int *x, int *y) {
    temp = *x;
    *x = *y;
    *y = temp;
}
```
Queues/Reentrancy
Reentrancy

- An easy fix in this case is to move the definition of temp inside the function.

- If you must call a non-reentrant function from a multi-tasking application you have several options:
  - Restrict all calls to the function to a single task.
  - Treat function calls to non-reentrant functions as critical sections. Protect with a mutex, disable preemption, etc.
Queues/Reentrancy

Reentrancy

- RTEMS uses the newlib version of the standard C libraries (libc and libm). Whenever possible the standard library functions are reentrant.

- For those standard routines that could not be made reentrant, a special reentrant version (with a different name) of the standard routine is provided. Refer to the newlib documentation (for libc and libm) at the newlib web site.
// Non-reentrant version of fopen()
#include <stdio.h>
FILE *datafile =
    fopen("datain.txt", "rb");
// Reentrant version: _fopen_r()
// A reentrancy structure is needed by
// each thread.
#include <stdio.h>
#include <reent.h>
struct _reent r;  // Must be local!!
FILE *datafile =
    _fopen_r(&r, "datain.txt", "rb");
Many of the stdio routines (scanf, getchar) are not reentrant. Although the special reentrant versions of these routines can be used by multiple tasks, a cleaner approach is to use a single task for all I/O. Message queues can be used to pass data between client tasks and the I/O server task.