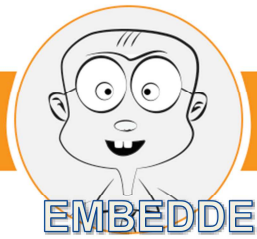


EMBEDDED SYSTEM COURSE

LECTURE 12: FREESCALE MQX RTOS TASK SYNCHRONIZATION

Learning Goals

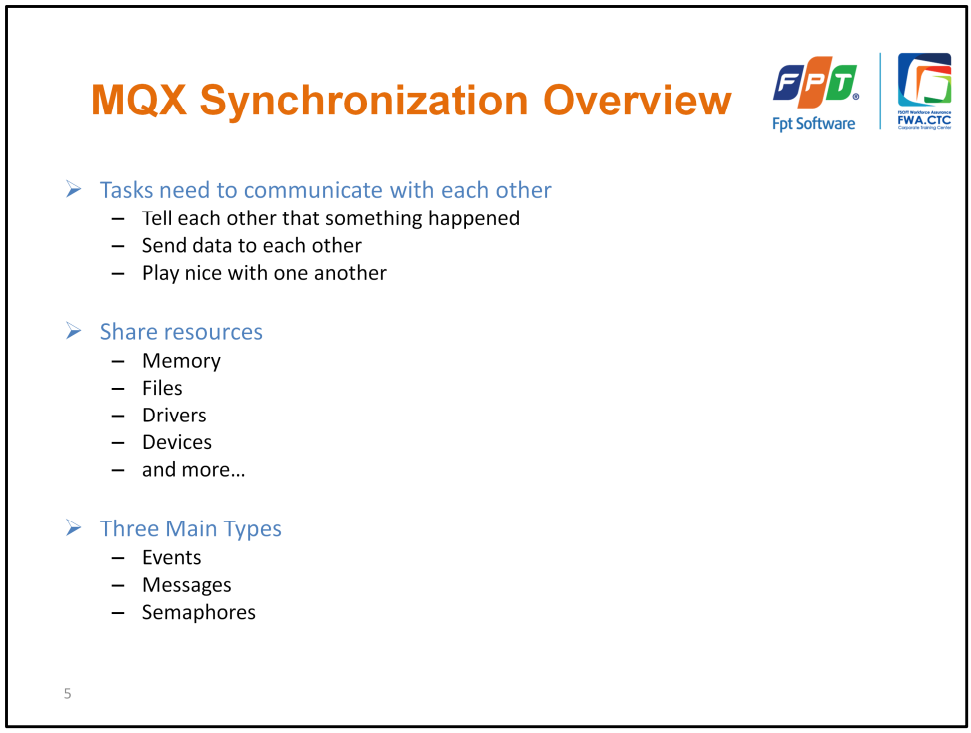
- Having deep understanding about the synchronization in MQX.
- Go through almost important components in MQX including event, lightweight event, message pool, message queues, etc.

2

- **MQX Synchronization Overview**
- **MQX Event**
  - Event Overview
  - Event Bits
  - Lightweight Events
  - Examples
- **MQX Message**
  - Message Structure
  - Message Pools
  - Message Queues
  - Sending and Receiving messages
  - Examples

Table of contents

Table of contents



Time_delay not exact, just guarentee the minimum amount of delay. Ie if say delay 7ms, will delay for 10ms since tha'ts smaller than BSP resolution

Table of contents

MQX Events

- Tasks can wait for a combination of event bits to become set. A task can set or clear a combination of event bits.
- Events can be used to synchronize a task with another task or with an ISR.
- The event component consists of event groups, which are groupings of event bits.
  - 32 event bits per group (mqx_uint)
- Tasks can wait for all or any set of event bits in an event group (with an optional timeout)
- Event groups can be identified by name or by index (fast event groups)

7

Tasks or ISR's can set event bits

If only identified by index, then a fast event group, since don't have to wait for name translation

Events

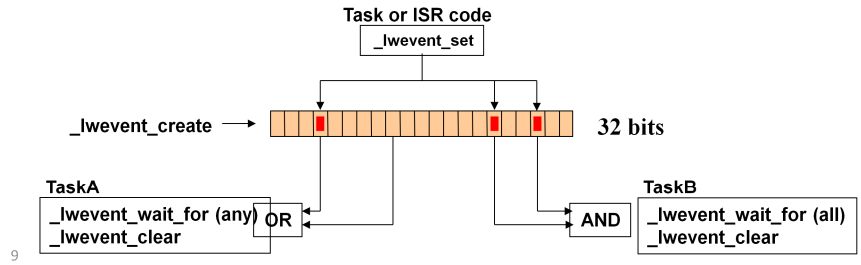
- Any task can wait for event bits in an event group.
- If the event bits are not set, the task blocks.
- When the event bits are set, MQX puts all waiting tasks, whose waiting condition is met, into the task's ready queue.
- If the event group has autoclearing event bits, MQX clears the event bits as soon as they are set
- Can use across processors (not possible with lightweight events)

8

With auto-clear, no need to clear bits manually

Event Bits

- A task waits for a pattern of event bits (a mask) in an event group with **_event_wait_all()** or **_event_wait_any()**.
  - Wait for all the bits in the mask provided to be set
  - Or wait for any of the bits
- When a bit is set, MQX makes ready the tasks that are waiting for the bit.
  - A task can set a pattern of event bits (a mask) in an event group with **_event_set()**.



9

When event group is created, that determines if it auto-clears or not

Full API in MQXRM and MQXUG

Light Weight Events

- Lightweight events are a simpler, low-overhead implementation of events.
- Lightweight event groups are created from static-data structures and are not multi-processor.
- To create a lightweight event group, an application declares a variable of type **LWEVENT_STRUCT**, and initializes it by calling **_lwevent_create()** with a pointer to the variable and a flag indicating, whether the event group has autoclearing event bits.

10

This compared with regular events which require creating the event component, then an event group, and then connecting to the event group

Most of the time LW event will do what you need it to. But regular event allows multi-processor events

Event Hands-on

- Create two tasks, one that sets the event bit and the other that waits for it

Table of contents

Messages

- Tasks can communicate with each other by exchanging messages
- Messages are areas of memory divided into a header and a data area
  - Application data is user-defined
  - Data area is of type MESSAGE_HEADER_STRUCT, which has these fields:
    - SIZE — size of the message (including this header) in SAUS
    - TARGET_QID — destination queue
    - SOURCE_QID — optional source queue
    - CONTROL — reserved for endian conversion
- Messages can be assigned a priority or marked urgent when sent

13

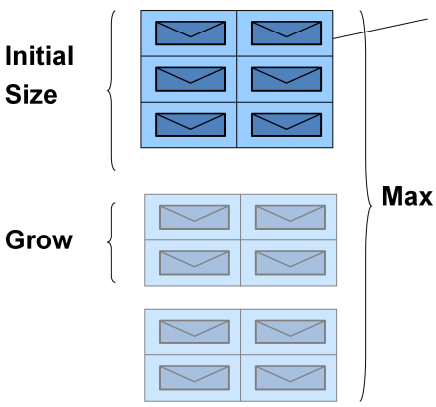
Envelope is message pool, and mailbox is message queue. Can use different sizes of "envelopes", so that's why can tell it how much memory to allocate for message pool

Talk about message pools in the next slide

Used as a synchronization mechanism across tasks. Can even communicate in tasks across processors. When look up message queue number, figures out how to route it if not on processor.

May need to set enable in user_config.h to use.

Message Pools

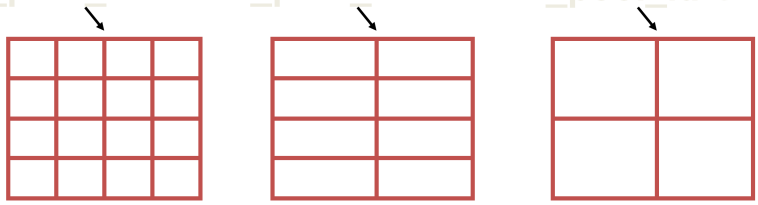


- A message pool is defined by:
  - Message size
  - Initial size
  - Grow amount
  - Max size

System Message Pools

- System pools are linked together
  - Messages are allocated on a first-fit basis
  - Easiest type of pool to use
  - Having different sized pools allows for more efficient use of memory, but longer allocation times
  - Created with `_msgpool_create_system()`

Private Message Pools



- Private pools are accessed individually via “pool ids”
  - Allows each part of application to use a different pool
  - Allows more control over how many messages are allocated
  - Create with `_msgpool_create()`

16

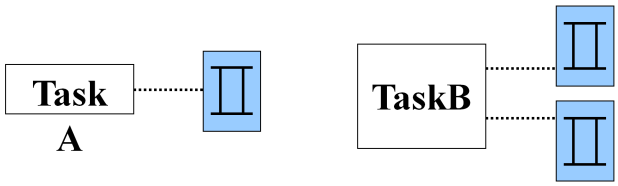
Allocating Messages

- All messages are obtained from message pools using a best-fit algorithm
- Messages are a resource of the task

17

Before a task sends a message, it allocates a message (_msg_alloc_system() or _msg_alloc()) of the
appropriate size from a system- or private-message pool.
System-message pools are not the resource of any task, and any task can allocate a message from them.
Any task with the pool ID can allocate a message from a private-message pool.
When a task allocates a message from either type of pool, the message becomes the resource of the task,
until the task frees the message (_msg_free()) or puts it in a message queue (_msgq_send family of
functions). When a task gets a message from a message queue (_msgq_poll() or _msgq_receive family),
the message becomes the resource of the task. Only the task that has the message as its resource can free
the message.

Message Queues



- Each task can have one (or more) messages queues associated with it
- Messages are always addressed to queues, not tasks
- Queues are identified by _queue_id
  - This is a combination of queue number and CPU number
- Create a queue using `_msgq_open()`

Message Queues

- Message queues 1-7 are reserved for system, so need to start message queue with 8.
- Tasks send messages to message queues, and receive messages from their message queues.
- System queues are not owned by a task and do not block when checked

19

Task can send to any private message queue, but can only receive on ones that it opened itself

Queue number starts with #8. 1-7 are reserved for internal MQX system

Sending Messages

- Messages are sent to message queues
- The send operation doesn't block
- To send a message (of lowest priority, 0):

```
boolean _msgq_send(msg_ptr)
```

20

Can use API to send messages with a specific priority or highest priority (0-15), 15 is highest priority

the message is placed at the head of the message queue, even if other urgent messages are in the queue

Broadcast msg send takes an array of queue ID's to send to if use that to send messages with

Send to any message queue.

Receiving Messages

- To receive a message (blocking) from a private message queue with msgq_receive()
- Non blocking with msgq_poll()
  - Used for ISR's
- A message should be freed after it's been received and processed
- To free a message that was allocated from either the system message pools or from a private message pool:

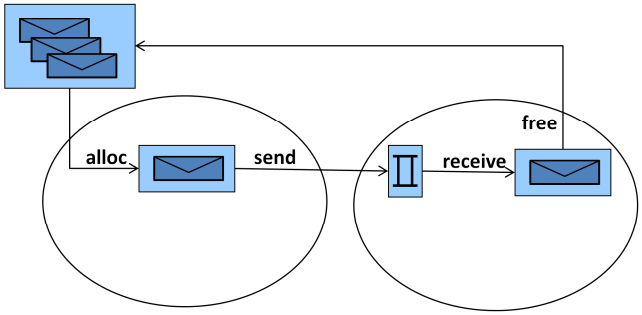
  `_msg_free`(*msg_ptr*)

21

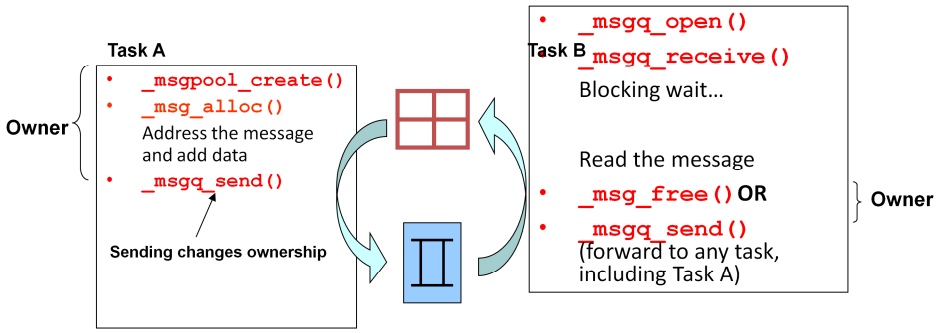
Remember that when a task recieves a message, it now becomes the owner of that message and is responsible for freeing it. Task can only receive on a messgae queue it created or else a system queue.

There are timeout options for receive queues so don't end up waiting forever
Can also get number of messages in queue with msg_get_count()

Simple message exchange



Message passing example



- **Message must "travel" in a loop:**
  - **Allocate it from a pool**
  - **Use it**
  - **Return it to pool (i.e. free it)**

Closing and Destroying Queues

- For the task that opened a private message queue to close it, or for any task to close a system message queue:

  `boolean` **`_msgq_close`**`(`*queue_id*`)`
- All messages in the queue are freed

- For any task to destroy a private message pool:

  `_mqx_uint` **`_msgpool_destroy`**`(`*pool_id*`)`
  - if all the messages in the message pool aren't first freed, the message pool isn't destroyed
- You can't destroy system message pools

Code

- Look in <mqx_install_dir>\mqx\source\message\ms_send.c
  - And ms_sendi.c

25

Also just look at example in MQX directory, would take too long to write it out. Walk through the example.

Table of contents

Semaphores

- MQX provides:
  - Lightweight Semaphores (LWSems)
  - Semaphores
  - Mutexes.

- Both types of semaphores can be used for task synchronization and mutual exclusion.

27

More about MQX interrupts later

Configure any signals that will not be using MQX

How semaphores work

- A semaphore has:
  - counter — maximum number of concurrent accesses
  - queue — for tasks that wait for access
- If a task waits for a semaphore

  if counter > 0

    counter is decremented by 1

    task gets the semaphore and can do work

  else

    task is put in the queue
- If a task releases (post) a semaphore

  if at least one task is in the semaphore queue

    appropriate task is readied, according to the queuing policy

  else

    counter is incremented by 1

28

This should be a quick review from the basic RTOS section earlier

Mutex is special case where the count =1 so only one task can hold it at a time

Semaphores

- Two types of semaphore classes
  - Strict – Task must have decremented (owned) the object it's posting
    - Counter is bounded by initial value
  - Non-strict – Any task can post the semaphore.
    - Counter is unbounded
- Uses for non-strict semaphores include:
  - if tasks always wait before posting, faster and lower-overhead non-strict semaphores are safe to use
  - if the semaphore is associated with a dynamically allocated resource, it may need to have its counter increased past its initial count

29

See if anyone can think of time when a non-strict semaphore would be used:

if tasks always wait before posting, faster and lower-overhead non-strict semaphores are safe to use
if the semaphore is associated with a dynamically allocated resource, it may need to have its counter increased past its initial count

Semaphore Queuing

- Three types of queuing
  - FIFO
    - Semaphore goes to the longest-waiting task.
  - Priority queuing
    - The queue of tasks waiting for the semaphore is in priority order, and MQX puts the semaphore to the highest-priority waiting task.
  - Spin
    - Spin for a time-sliced amount of time. Not recommended

30

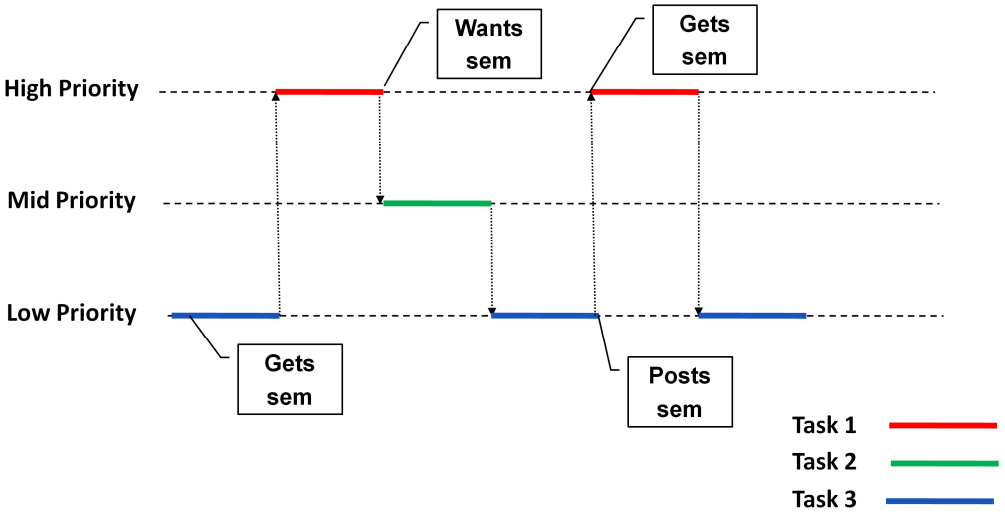
Priority Inversion

- Occurs when a low-priority task causes a higher-priority task to block
- See diagram

31

Draw diagram on whiteboard showing hypotehical situation. See if can figure out what the problem is after P1 tries to get the mutex

Priority inversion illustrated



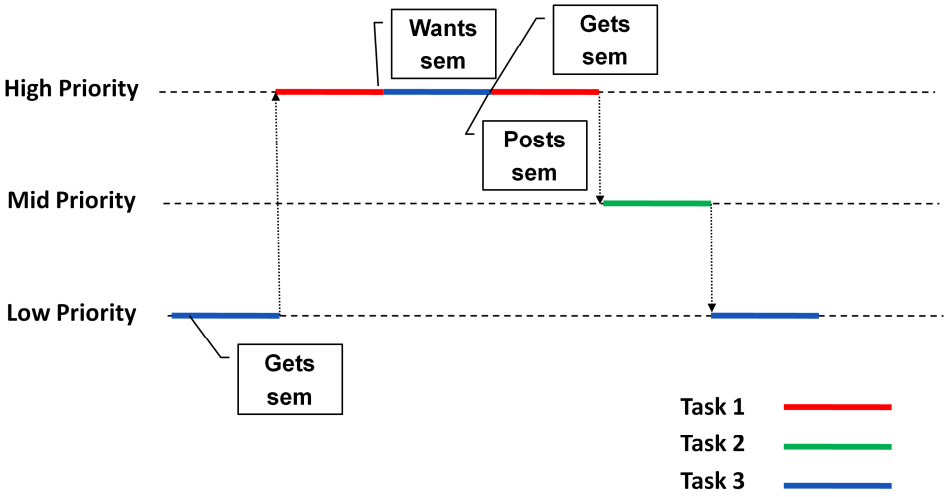
Priority Inheritence

- Fix Priority Inversion two ways:
  - Priority Inheritence
  - Priority Protection
- Priority Inheritence
  - If a higher-priority task waits for the semaphore or mutex, MQX temporarily raises the priority of the task that has the semaphore or mutex to the priority of the waiting task.
  - The task that has the mutex can finish it's work, and then release the mutex
  - And then drops back down to low priority, allowing the high priority task to take mutex and start it's own work

33

Draw this on the whiteboard

Priority inheritance illustrated



34

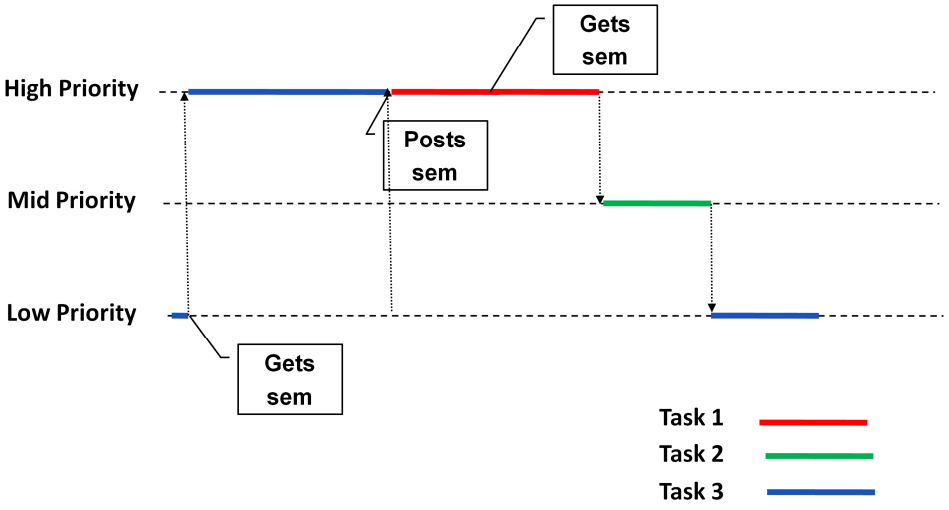
Priority Protection

- Task's priority is increased to that of the mutex-specified priority until the task unlocks the mutex
- Never gets pre-emptied by a task with a priority lower than the mutex

35

Draw on board

Priority protection illustrated



36

Lightweight Semaphore

- These are a core component
- Created from static-data structures, and are not multi-processor

- To create a lightweight semaphore, you declare a variable of type **LWSEM_STRUCT,** and initialize it by calling **_lwsem_create()** with a pointer to the variable and an initial semaphore count.

- Semaphore count indicates the number of requests that can be concurrently granted the lightweight semaphore
  - Remember that the counter is unbounded since not strict

37

LW Semaphores

- _lwsem_wait() to try and decrement
- _lwsem_post() to increment

- Uses FIFO queuing
- If tasks at different priorities access the same semaphore, semaphores should be used rather than a lightweight semaphore
  - avoids priority inversion

- Can also poll with lwsem_poll()
  - Non-blocking, so can use in ISR's

38

Use lwsem_poll if want to check status of a semaphore to decide what to do, but without waiting for it

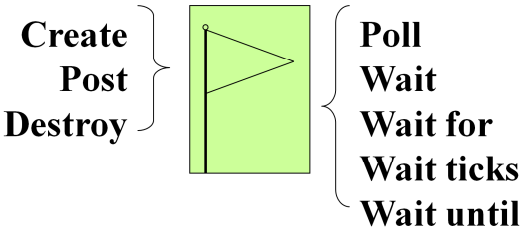
Semaphore

- Upon creation
  - Sets up initial count
  - Strictness
  - Priority Inheritance
  - Priority queue or FIFO
- sem_post() and sem_wait() to use semaphore
  - Can wait only so long if desired
  - See se_wait.c, se_waiti.c, and se_post.c in the sem directory for code

39

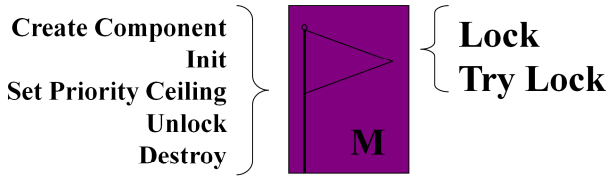
If use priority inheritance (task prioirty only raised when have semaphore), then must be strict

Lightweight Semaphore Overview



- MQX Core Component
- Useful for:
  - Task Synchronization
  - Count of available resources
- Not strict – count is un-bounded

Overview of MQX Mutexes



- MQX Optional Component
- Attributes control operation
- Identified by address (similar to lightweight events)
- No limit to the number of mutexes

Semaphore Summary

	LW Sem	Semaphore	Mutex
Timeout	Yes	Yes	No
Queuing	FIFO	FIFO, priority	FIFO, priority, spin
Strict	No	No or yes	Yes
Priority inheritance	No	Yes	Yes
Priority protection	No	No	Yes
Size	Smallest	Largest	Mid
Speed	Fastest	Slowest	Mid

Exercises

- Write an application that has two tasks, which task will operate a certain led. Each led must be blinked sequencely.
- Write an application that has three tasks: Producer1, Producer2, Consumer. Using message queue for Producer1 and Producer2 can send a message to Consumer. In Consumer task, each message will be printed out to console one by one.

Summary

- Understanding about the basic concepts regarding task synchronization on MQX RTOS

Question & Answer

Thanks for your attention !

Copyright

- This course including **Lecture Presentations**, **Quiz**, **Mock Project**, **Syllabus**, **Assignments**, **Answers** are copyright by FPT Software Corporation.
- This course also uses some information from external sources and non-confidential training document from Freescale, those materials comply with the original source licenses.