GST[050.HCM.UIT]

Name: Nguyen Duc Phu

ID: 12520671

Freescale MQX RTOS

❖ Architecture of the RTOS

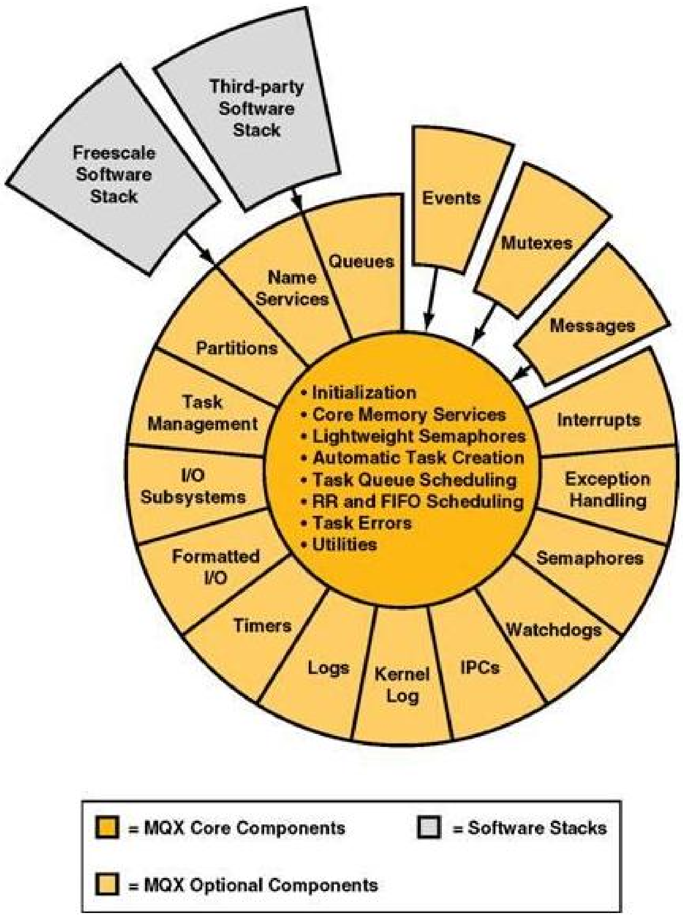


Figure 1: The modular, component – based architecture of the MQX RTOS

- Fully functional version with support for major platforms Freescale hardware (Kinetis, Vybrid, ColdFire and Power Architecture).
- Extremely configurable, divided into 25 components (8 essential and 17 optional).
- Drivers compatible with the POSIX standard.
- Consume program memory 6K (ROM) and 2.5K of memory data (RAM) in a typical configuration with an ARM Cortex-M4.

Features of an RTOS:

- Allows multi-tasking
- Scheduling of the tasks with priorities
- Synchronization of the resource access
- Inter-task communication
- Time predictable
- Interrupt handling

Plan to use drivers that are available with an RTOS

- Would like to spend your time developing application code and not creating or maintaining a scheduling system
- Multi-thread support with synchronization
- Portability of application code to other CPUs
- Resource handling
- Add new features without affecting higher priority functions
- Support for upper layer protocols such as:
    - TCP/IP, USB, Flash Systems, Web Servers,
    - CAN protocols, Embedded GUI, SSL, SNMP

❖ What is the pros/cons of this RTOS?

Key Benefits

- **Small code density –** The Freescale MQX RTOS can be configured to take as little as 8 KB of ROM and 2.5K RAM on ARM Cortex M4, including kernel, 2 task applications,1 LW Semaphore, interrupt stack, queues, and memory manager.
- **Component-based architecture –** Provides a fully-functional RTOS core with additional, optional services. Components are linked in only if needed, preventing unused functions from bloating the memory footprint.
- **Full and lightweight components –** Key components are included in both full and lightweight versions for further control of size, RAM/ROM utilization and performance options.
- **Real-time, Priority-based preemptive, multithreading –** Priority-based preemptive scheduling allows high-priority threads to meet their deadlines consistently, no matter how many other threads are competing for CPU time.
- **Optimized for Freescale architecture –** Optimized assembly code to accelerates key real-time portions of the RTOS such as context switching.
- **Scheduling –** Freescale MQX RTOS provides the developer faster development time by relieving engineers from create or maintain an efficient scheduling system and interrupt handling. It is also significantly useful if one requires the use of multiple communication protocols like USB or TCP/IP.

- **Code Reuse –** Freescale MQX RTOS provides a framework with a simple API to build and organize the features across Freescales broad portfolio of embedded processors.

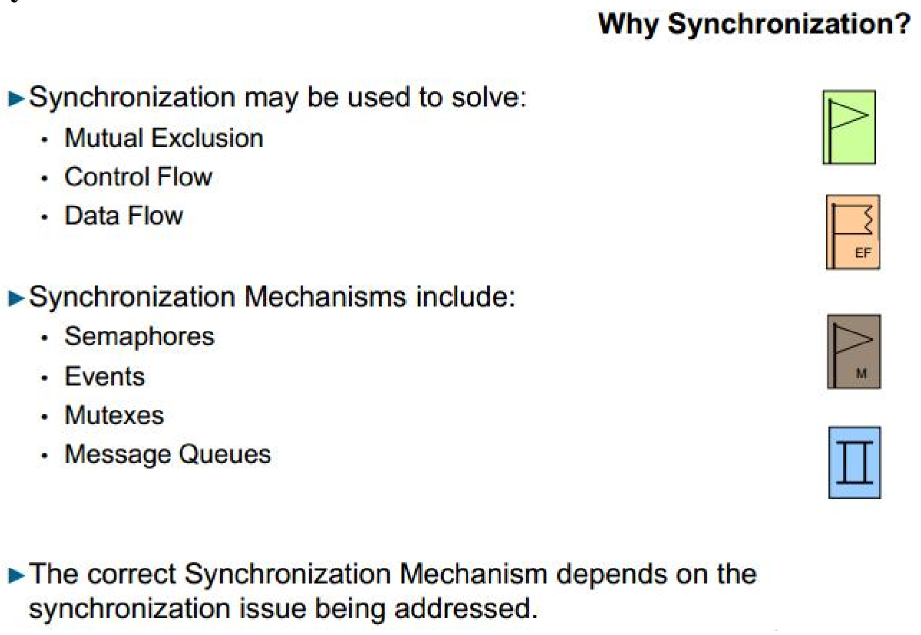
- **Intuitive API –** Writing code for Freescale MQX RTOS is straight forward with a complete API and available reference documentation.
- **Fast boot sequence –** A fast boot sequence ensures the application is running quickly after the hardware has been reset
- **Simple Message Passing –** Messages can be easily passed between tasks running on the same CPU or even a different CPU in the system.

Limitations

- An RTOS makes it much easier to develop systems real-time characteristics, but it can not be the solution ideal for a given project:
    - The services provided by the kernel add an overhead implementation, which may vary between 2% and 5% CPU usage, depending the RTOS.
    - A RTOS needs an extra space ROM (Flash) to store code, which may vary from a few hundred bytes up to a few hundred kilobytes.
    - A RTOS consumes RAM to store the context and the stack of each task, which can vary from a few hundred bytes to tens of kilobytes
- Expensive
- Use many system resources
- The algorithm is complicated.

❖ **Real application in the market:** in embedded system, modem, medical equipment, military, trading,…

❖ **Synchronization in this RTOS**



Why Synchronization?

► Synchronization may be used to solve:
- Mutual Exclusion
- Control Flow
- Data Flow

► Synchronization Mechanisms include:
- Semaphores
- Events
- Mutexes
- Message Queues

► The correct Synchronization Mechanism depends on the synchronization issue being addressed.

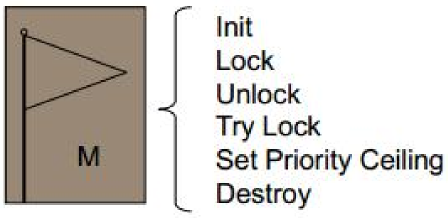
- Control Flow provides a mechanism for a task or ISR to resume execution of one or more other tasks:
    - Mutual exclusion is used to prevent another task from running
    - Control Flow is used to allow another task to run
- Not all synchronization objects are suitable for control flow:
    - Good: Non-strict Semaphores, Events, Messages, Task Qs
    - Bad: Mutexes, Strict Semaphores
- Data Flow provides a mechanism for a task or ISR to resume execution of one or more other tasks with data
- Mutual exclusion is not necessary
- Messages are the primary means of data flow synchronization

Attributes of MQX Synchronization Mechanisms

- Reference by name vs. by address
- Strict
- One-to-One vs. One-to-Many
- Convey information
- Priority queuing
- Priority inheritance
- Multi-processor capable
- Ownership tracking

1. **MQX Mutexes:**
   Mutexes are used for mutual exclusion, so that only one task at a time uses a shared resource such as data or a device. To access the shared resource, a task locks the mutex associated with the resource. The task owns the mutex, until it unlocks the mutex.

   MQX Optional Component
   - Attributes control operation
   - Identified by address
   - No limit to the number of mutexes



2. **Semaphores:**

A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:

- o control access to a shared resource (mutual exclusion)
- o signal the occurrence of an event
- o allow two tasks to synchronize their activities

A semaphore is a ―token‖ that your code acquires in order to continue execution.

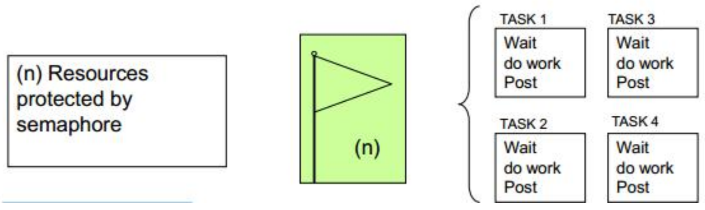
- o If the semaphore is already in use, the requesting task is suspended
- o until the semaphore is released by its current owner.

A semaphore has:

- o counter — maximum number of concurrent accesses
- o 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



A semaphore has:

- o Counter – maximum number of concurrent accesses (n)
- o Queue – for tasks that wait for access

If a task waits for a semaphore

- o If n>0, n is decremented and task runs

- If n is 0, task is put on the waiting queue

When a task releases (posts to) a semaphore

- If a task is in the queue, task is readied per the queuing policy
- No queued tasks, n is incremented

If a semaphore is strict, a task can‘t release (post) it unless the task first waited for and got the semaphore

- the counter is bounded by its initial value

If a semaphore isn‘t strict, a task can release (post) it without
first waiting for it

- the counter is unbounded

Uses for non-strict semaphores include:

- if tasks always wait before posting, faster and lower-overhead non-strict emaphores 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.

3. **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_unit)
- 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)
- 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 events across processors (not possible with lightweight events.)
- A task waits for a pattern of event bits (a mask) in an event group with _event_wait_all() or _event_wait_any().

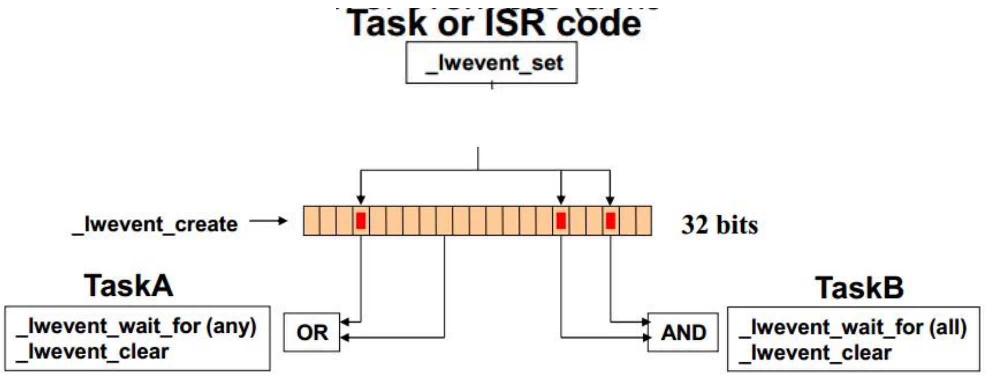
Wait for all the bits in the mask provided to be set

or

Wait for any of the bits

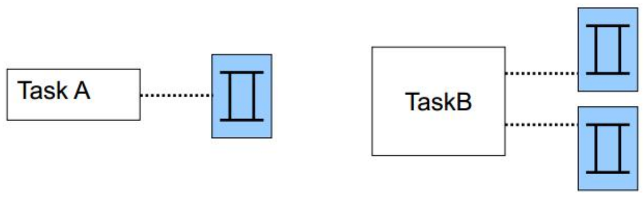
- When a bit is set, MQX makes ready the tasks that are waiting for the bit.

- A task can set a pattern of event bits (a mask) in an event group with _event_set().



4. Message passing:

- Tasks can communicate with each other by exchanging messages.
- Tasks send messages to message queues, and receive messages from message queues.
- Messages can be assigned a priority or marked urgent.
- Tasks can send broadcast messages
- Messages are areas of memory divided into a header and a data area
- Application data is user-defined
- Message Pools are allocated from core memory
- A message pool is defined by:
  - Message size
  - Initial size
  - Grow amount
  - Max size
- They contain several equal-sized messages
- Benefit is fast allocation/de-allocation and no fragmentation
- Each pool can contain different size messages
- System can automatically add new messages to a pool if it fills up (i.e. all messages are in use)
- 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 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()

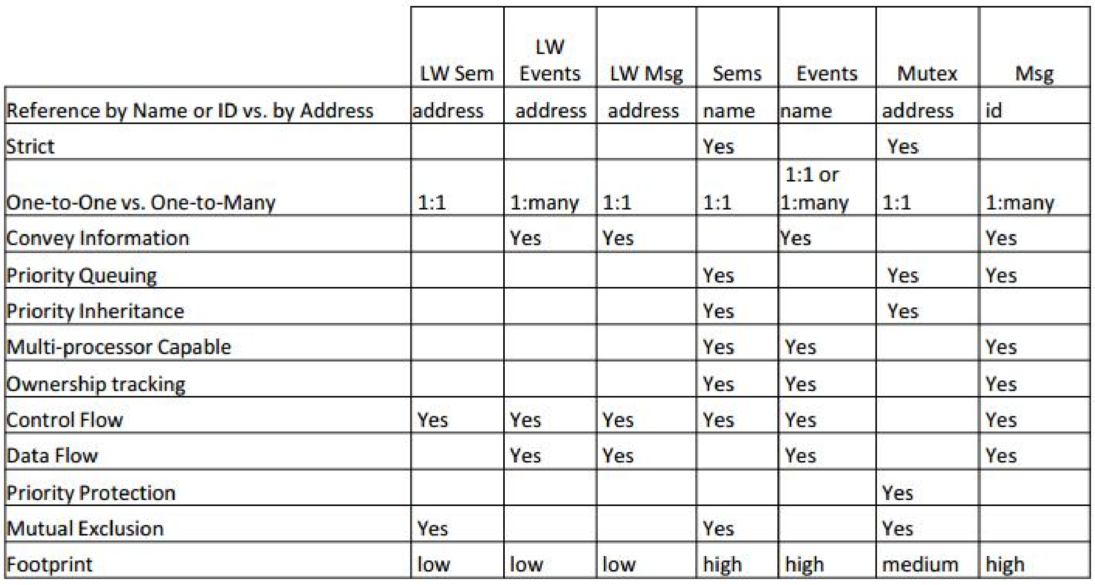


- Each task can have one (or more) messages queues associated with it
- Messages are always addressed to queues, not tasks
- Queues are identified by _queue_id

  •This is a combination of queue number and CPU number

- Create a queue using _msgq_open()

Attributes of MQX Synchronization Mechanisms



	LW Sem	LW Events	LW Msg	Sems	Events	Mutex	Msg
Reference by Name or ID vs. by Address	address	address	address	name	name	address	id
Strict				Yes		Yes	
One-to-One vs. One-to-Many	1:1	1:many	1:1	1:1	1:1 or 1:many	1:1	1:many
Convey Information		Yes	Yes		Yes		Yes
Priority Queuing				Yes		Yes	Yes
Priority Inheritance				Yes		Yes	
Multi-processor Capable				Yes	Yes		Yes
Ownership tracking				Yes	Yes		Yes
Control Flow	Yes	Yes	Yes	Yes	Yes		Yes
Data Flow		Yes	Yes		Yes		Yes
Priority Protection						Yes	
Mutual Exclusion	Yes			Yes		Yes	
Footprint	low	low	low	high	high	medium	high

Hết! ☺