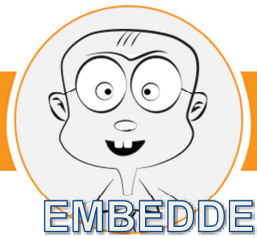


EMBEDDED SYSTEM COURSE

# LECTURE 10: INTRODUCE TO BASIC REAL-TIME APPLICATIONS AND RTOS

# Learning Goals

- Overview on Freescale MQX RTOS.
- Introduce and instructs on creating a task in MQX RTOS.
- Introduce  about scheduling in MQX RTOS.
- Introduce about task synchronization in MQX RTOS.

2

- Freescale MQX Overview
- MQX Basics: Tasks
  - Hands-on
- MQX Basics: Scheduling
  - Hands-on
- MQX Basics: Task Synchronization
  - Semaphores
  - Hands-on
- Additional Resources
- Review

# Table of contents

# Table of contents

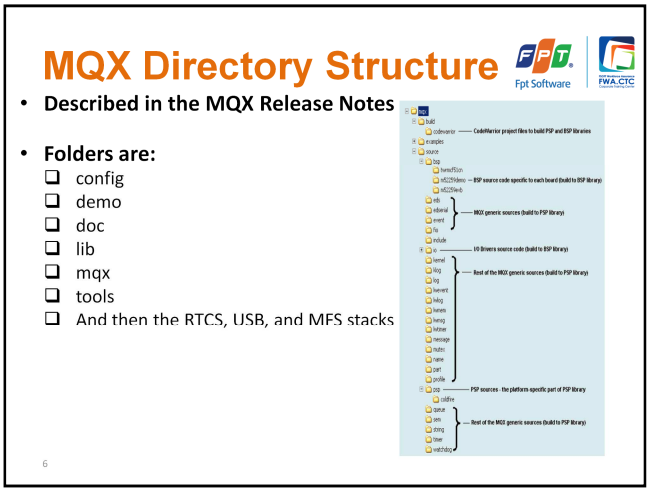
# Freescale MQX

- We will be using Freescale MQX to demonstrate these RTOS concepts
- Freescale MQX Software can be downloaded:
  - http://www.freescale.com/mqx
- Default Freescale MQX folder:
  - C:\Program Files\Freescale\Freescale MQX 3.7

5

MQX has already been installed on your laptops. But if you want to check it out after the class, you can go to http://freescale.com/mqx to download a copy. It is also included on the DVD that comes with the MCF51CN tower module, but the latest version will be on the web.

The default installation directory is in C:\Program Files\Freescale\Freescale MQX 3.5
::Show your Windows Explorer to that directory location::




The first is the config directory. You'll see that it contains all the different hardware platforms that are supported by MQX, and inside each of those folders is a user_config.h file that contains the configuration settings for that board. Most changes to MQX configuration take place in that file.

You'll also see a folder for the CodeWarrior 7.2 compiler, which contains a project file for re-compiling the MQX libraries. If any changes are made to the user_config.h file, the libraries must be re-compiled.

Going back up to the MQX root directory, the next folder is the demo folder. This contains the out of box demos and full-featured example code.

The next folder is the documentation folder. Inside you'll see the MQX release notes, which contain a lot of very useful information on the example projects available, what drivers and stacks are supported by each particular board, and what's new in this version of MQX.

A little deeper down you'll find the reference manuals for the different stacks, as well as the MQX user's guide. This document provides a good summary of how to use the MQX kernel and the features available.

The next folder to look at is the library folder. Looking in there, you'll see the pre-compiled libraries for each board that MQX supports. The files in this folder should not be edited, as they will be overwritten the next time the libraries are compiled. However this is the path your application will look for when calling the mqx function calls.

# MQX Directory Structure

- **The “mqx” directory is heart of MQX**
- Buid
  - Bat file
  - Codewarrior projects
- Examples
  - Examples1,2,3..
- Source
  - Include
  - IO, API source
  - Bsp
  - PSP…

## Changing Configuration Options

- User configuration options are set in

```
<mqx_install>/config/<board>/user_config.h
```

- Change the default serial port to UART1:

```
#define BSP_DEFAULT_IO_CHANNEL   "ttyb:"
#define BSP_DEFAULT_IO_CHANNEL_DEFINED
#define BSPCFG_ENABLE_TTYB       1
```

- Always need to re-build all the libraries with this new configuration

8

user_config.h header is the top most header file for the RTOS and its associated components. This header is used to over-ride default FSLMQX settings using #defines.

# Table of contents

# MQX RTOS Tasks

- A system consists of multiple tasks
- Tasks take turns running
- Only one task is active (has the processor) at any given time
- MQX manages how the tasks share the processor (context switching)
- Task Context
  - Data structure stored for each task, including registers and a list of owned resources

10

To start things off, lets think about how we solve problems as engineers. You have a big problem to solve. In order to make it more manageable, you then break it down into little problems to tackle individually.

It is the same idea when using an RTOS. You have a software problem you're trying to solve, and so you break it down into chunks, which are called tasks. An RTOS system consists of one or more tasks that work together to solve a larger problem.

The tasks takes turns running, as only one task, called the active task, has the processor at any one time. The RTOS then uses what is called a scheduler to determine how those tasks share the processor. And when a new task takes control of the processor, that's called a context switch.

By performing many context switches quickly, you can create the illusion of concurrency. However only task is active at a time.

# Typical Task Coding Structure

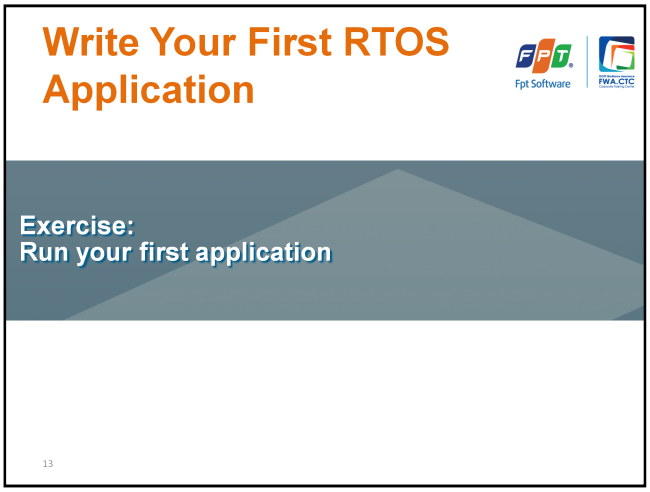
```
void mytask(uint_32 startup_parameter)  {
   /* Task initialization code */
   ....
   while (1) {
      /* Task body */
      ....
      ....
   }

}
```

11

A task may be instantiated more than once.

# Exercise




**The Exercise includes 2 tasks**

- Main Task:
    - AUTOSTART, priority 9
    - Prints out welcome
    - Loops, incrementing x
- Print Task:
    - AUTOSTART, priority 10
    - Loops, printing the value of x

**Follow the directions in Lab 1 to create, compile, and run the lab using Keil uVision**

**Pause the execution to see the main task running**

**When does the print task run?**

## Hands On: Run your first application

- The application has only one task – the Init Task
  - Prints out welcome
  - Loops, incrementing x
- Follow the directions in **Lab 1** to compile and run the lab using Keil uVision
- Pause the execution to see the init task running

## Task States

- **A task is in one of these logical states:**
  - ❑ blocked
    - the task is blocked and therefore not ready
    - it's waiting for a condition to be true
  - ❑ active
    - the task is ready and is running because it's the highest-priority ready task
  - ❑ ready
    - the task is ready, but it's not running because it isn't the highest-priority ready task
  - ❑ terminated
    - the task has finished all its work, or was explicitly destroyed

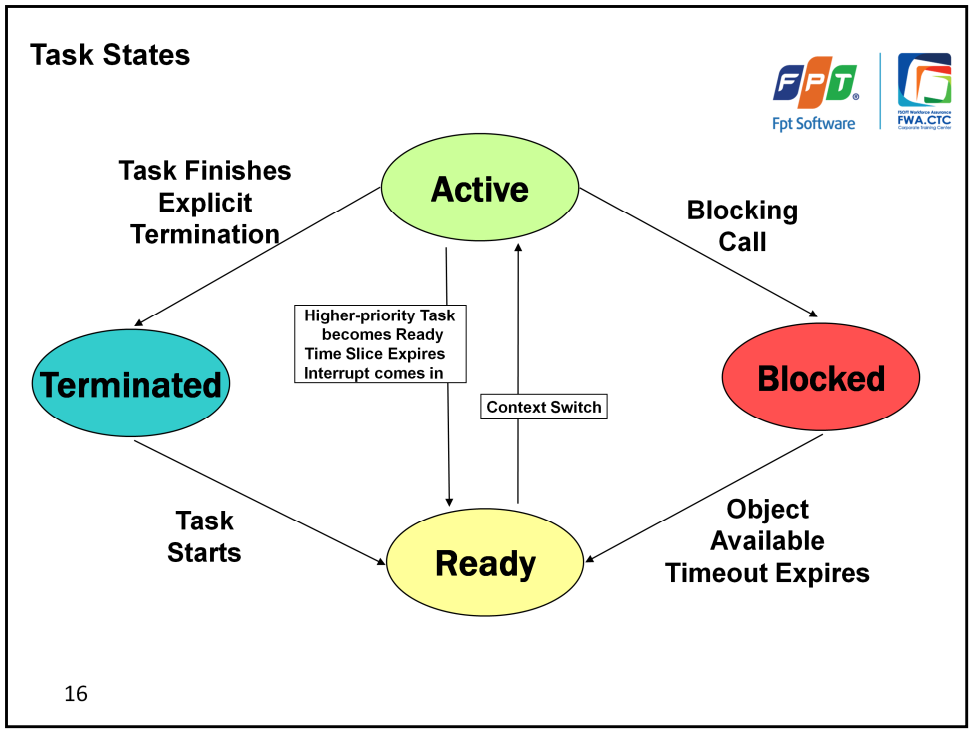
15

Tasks are the key building block of an RTOS, so lets look at the logical states that a task can be in.

The first state is a blocked. This means that the task is not ready and it is waiting on something. It could be waiting on a signal to occur, a timeout to expire, or a condition to become true.

Another state a task can be in is the active state. That means the task is ready, and it is the one that currently has control of the processor because it is the highest priority ready task.
The ready state happens when a task could run, but is not the active task because it is not the highest priority task.

Finally a task can be in the terminated state. This means a task has finished all its work and no longer exists, or it was explicitly destroyed.




This diagram shows the visual representation of the different task states. Looking at the active task, it can go to the terminated state if the task finishes its work or becomes explicitly terminated.

If it calls a blocking call, then it goes into the blocked state. This means it is now waiting for some condition to become true.
Finally if a higher priority task becomes ready, its time slice expires, or an interrupt occurs, then the active tasks gets put into ready state. At that point a context switch occurs, and a task that was in the ready queue gets promoted to being the active task.

If a task becomes unblocked because that condition became true, then it gets put into the ready state and the schedule determines if it is the highest priority task that could be running, and does a context switch if that is the case. Also when a task is first created, it gets put into the ready state.

# Table of contents

# Priorities

- Each task is assigned a priority
  - Higher number means lower priority
  - 0 is highest priority
- Used by scheduler to determine which task to run next
- User tasks should run at priority 8 or higher.

18

We've mentioned priorities several times, and in MQX, smaller numbers have a higher priority than a larger number. Thus a task with its priority set to 10 is a higher priority than a task set at 11. The priority levels should also be close together, so avoid the scenario where one task has a priority of 10 and the other is set to 100.

User tasks in MQX should start at priority 8, as the kernel and stacks run at higher priorities.

## Scheduler

- **Common Scheduling Configurations:**
  - ❑ FIFO (also called priority-based preemptive)
    - The active task is the highest-priority task that has been ready the longest
  - ❑ Round Robin
    - The active task is the highest-priority task that has been ready the longest without consuming its time slice

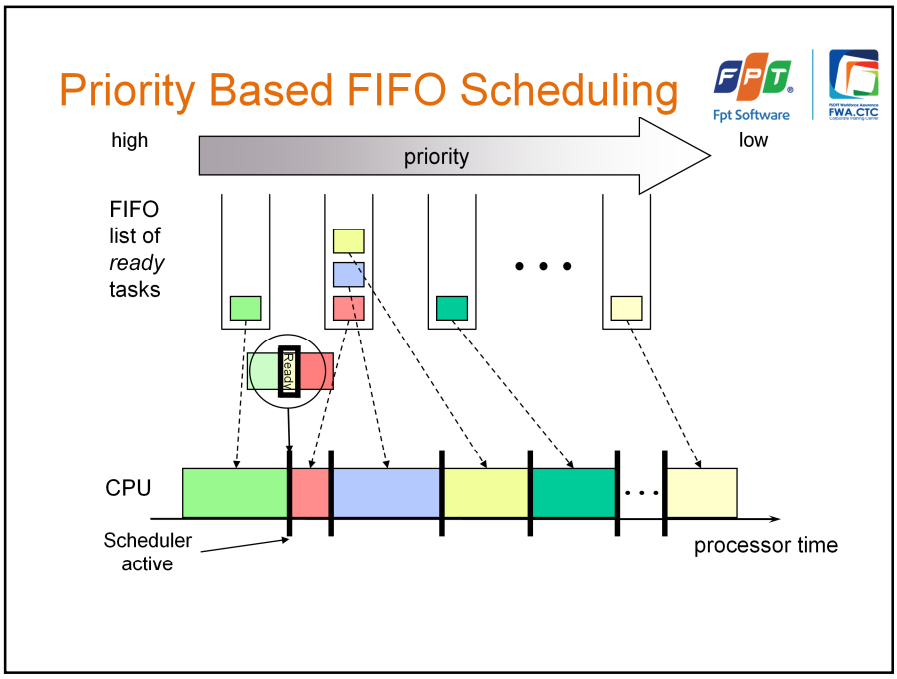
19

How does MQX determine which of the ready tasks gets to become the active task? By using a scheduler.

There are several scheduling policies that MQX supports. The most commonly used is First In-First Out. This means the task that becomes active is the highest priority task that has been ready the longest. A task will continue running as long as it is the highest priority ready task.

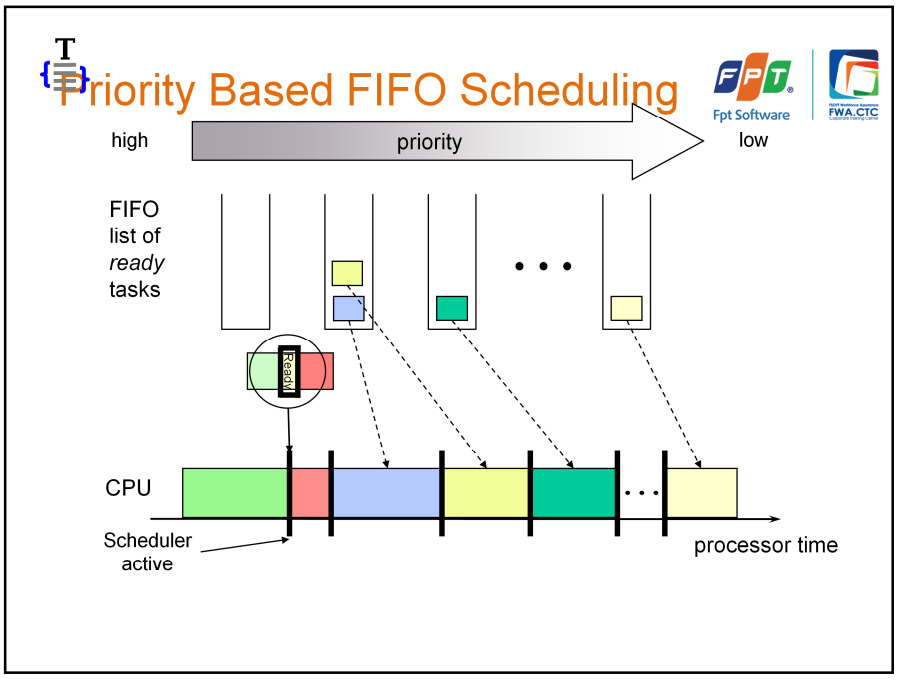
The next scheduling policy is called Round Robin. This behaves exactly the same as First In-First Out, except that the scheduler is explicitly called after a specified period of time, called a time slice. This allows other tasks at the same priority level to gain control of the processor. The length of the time slice can be set on a per-task basis, so not all tasks necessarily have to use round-robin, nor have the same time slice value.

Finally there is a way to explicitly schedule the tasks using task queues in MQX, but it is not used very often and will not be covered here



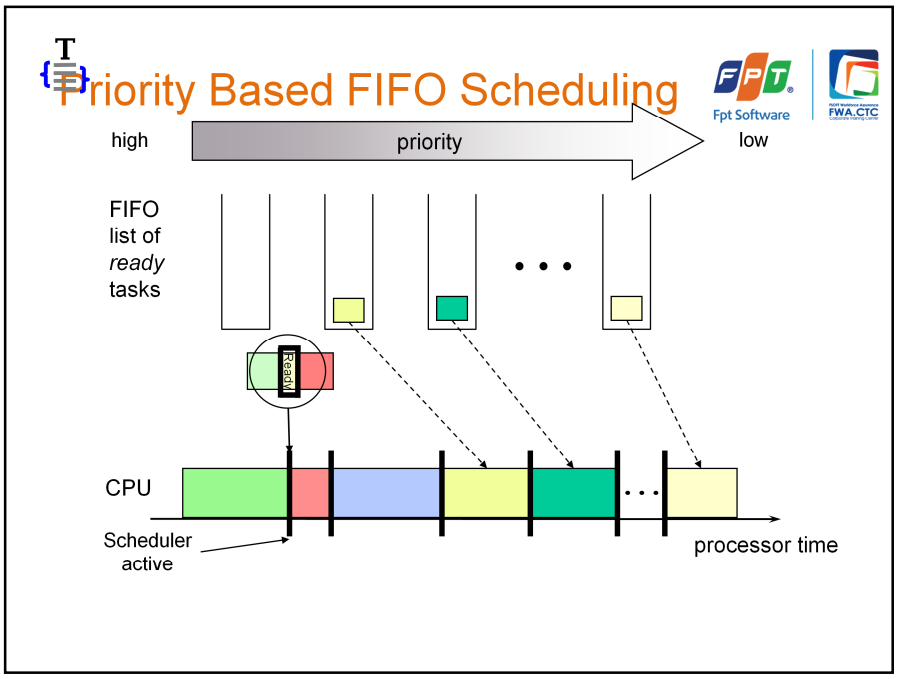

This diagram shows First In-First Out scheduling. Each priority has a queue associated with it, representing tasks was placed into the ready state at that priority level. The schedule first looks at the highest priority ready queue, and runs that task. That task will run as long as it needs until the task finishes.

Once the green task is finished, the scheduler is called and it looks at the highest priority queue. That queue is now empty, so it moves down to the next highest priority queue. The red task then runs until it is finished. The scheduler is called again, and it takes the next ready task that has been waiting the longest, which is the blue task. This continues on with the yellow task and so on until all the ready queues are empty.



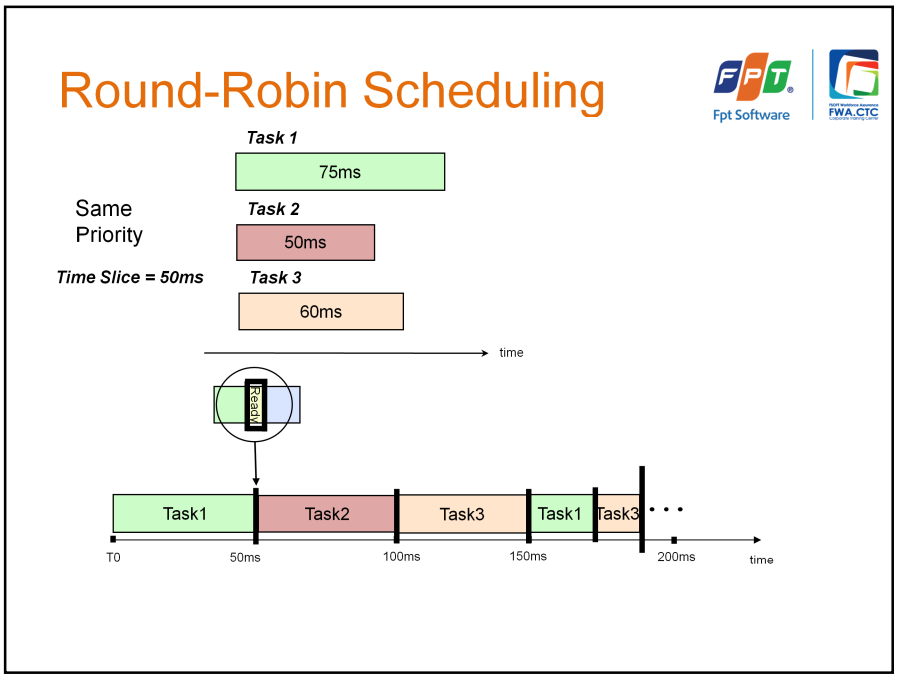

FIFO (also called priority-based preemptive) scheduling is a core component — the active task is the highest-priority task that has been ready the longest. FIFO is the default scheduling in Freescale MQX.

The active task runs, until any of the
following occurs:
-The active task voluntarily relinquishes the processor, because it calls a blocking MQX function.
-An interrupt occurs that has higher priority than the active task.
-A task that has priority higher than the active task, becomes ready.




FIFO (also called priority-based preemptive) scheduling is a core component — the active task is the highest-priority task that has been ready the longest. FIFO is the default scheduling in Freescale MQX.

The active task runs, until any of the
following occurs:
-The active task voluntarily relinquishes the processor, because it calls a blocking MQX function.
-An interrupt occurs that has higher priority than the active task.
-A task that has priority higher than the active task, becomes ready.

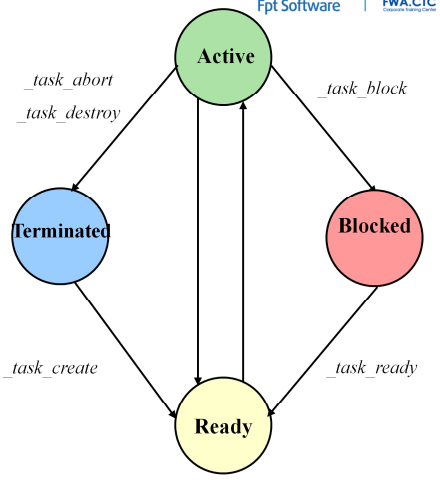



Round Robin scheduling works similarly. We have 3 tasks all with the same priority, but each take different amounts of time to finish executing their code.

Task1 runs first, but because the timeslice is set to 50ms, the scheduler is called before Task1 can finish. Task 2 then gets to run, and is able to finish in the 50ms time slice. Then task 3 becomes the active task (as it has been waiting the longest) and also runs for 50ms before being interrupted by the scheduler. Now task1 gets to run again until it completes its work, and then the scheduler is called one last time to let task3 become the active task until it also finishes its work.

# MQX Tasks

- Tasks can be automatically created when MQX Starts; also, any task can create another task by calling `_task_create()` or `_task_create_blocked()`
- The function `_task_create()` puts the child task in the ready state and the scheduler puts the higher priority task to run
- If `_task_create_blocked` is used the task is not ready until `_task_ready()` is called

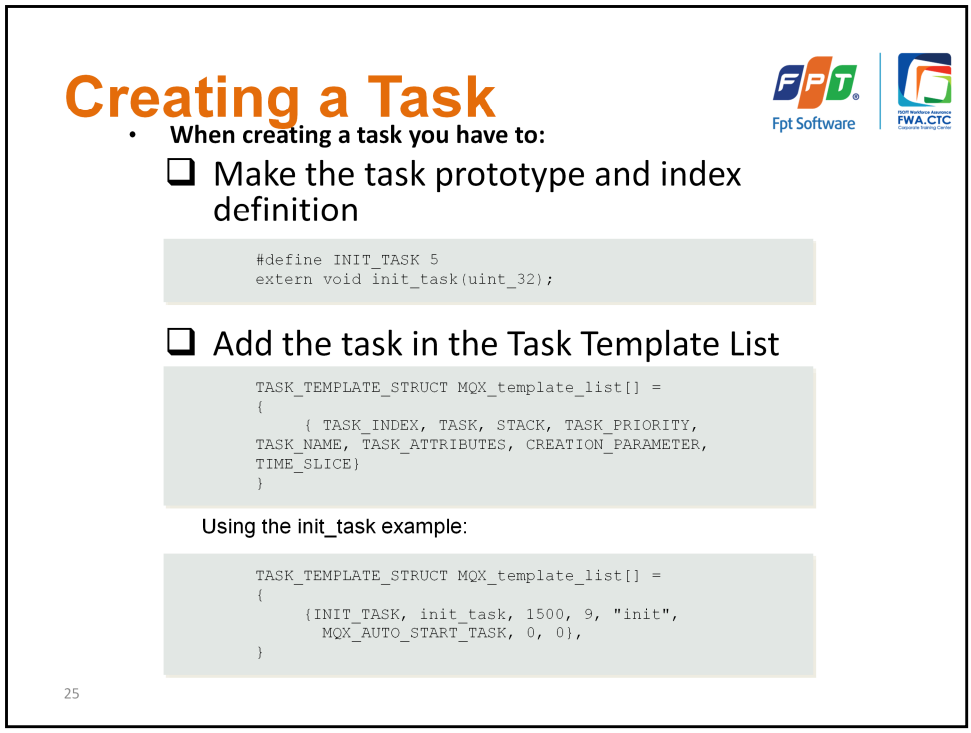



24

Now that we've covered the concept of tasks in an RTOS, lets link it with specific MQX calls. Tasks can be created when MQX starts, and also by calling the task_create function.

The active task can be put into the blocked state if it calls task_block on itself. Another task can then call task_ready on the blocked task to put it back into the ready state. And the active task can also end itself or any other task by calling task_abort or task_destory.

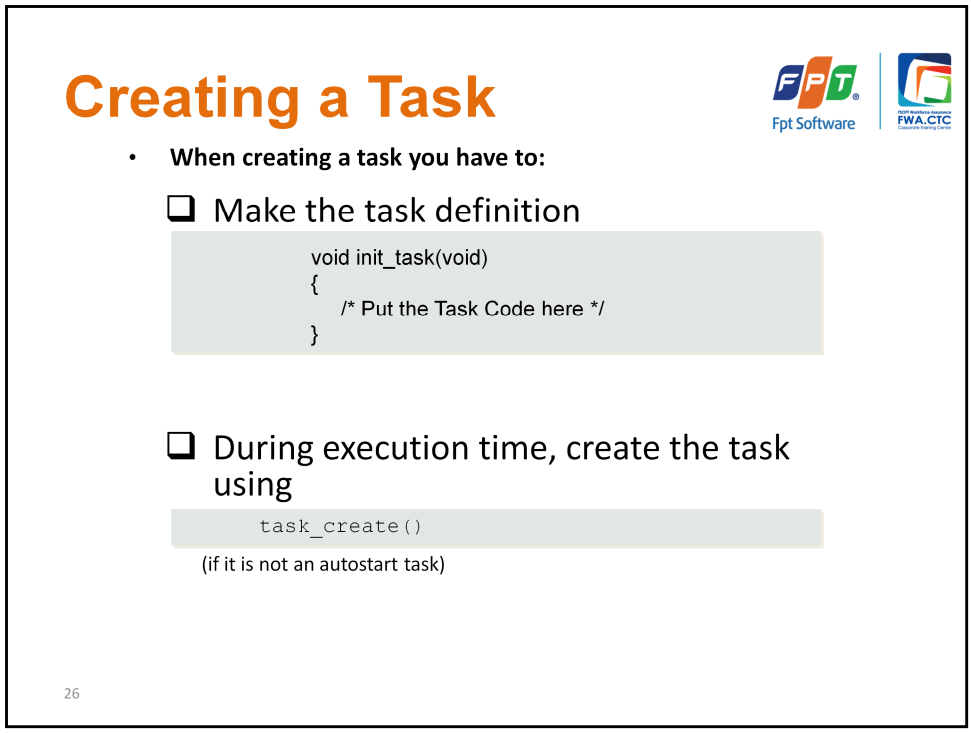
There are many other ways that a task can enter or leave these states. However this diagram shows some of the explicit commands that can be used to manipulate the task states directly.




In the Task Template Structure:
TASK_INDEX is usually a Define with an index number (other than the priority) of each task
TASK refers to the function name; the documentation calls it task address pointer but when using the task name C takes the address of the function
STACK is the defines stack size.
TASK_PRIORITY; the lower number, the higher priority. Task with priority 0 disables all the interrupts and priorities 0 to 8 are used by the OS Kernel
TASK_NAME is a string that helps to identify the task. It is also used to get the task ID.
TASK_ATTRIBUTES. You can use more than one attribute for each task on the list. The allowed Task attributes are:

- Auto start — when MQX starts, it creates one instance of the task.
- DSP — MQX saves the DSP co-processor registers as part of the task's context.
- Floating point — MQX saves floating-point registers as part of the task's context.
- Time slice — MQX uses round robin scheduling for the task (the default is FIFO scheduling).

CREATION_PARAMETER is the parameter to be passed to the task when it is created.
TIME_SLICE. Time slice (in milliseconds usually) used for the task when using round-robin scheduling.




In the Task Template Structure:
TASK_INDEX is usually a Define with an index number (other than the priority) of each task
TASK refers to the function name; the documentation calls it task address pointer but when using the task name C takes the address of the function
STACK is the defines stack size.
TASK_PRIORITY; the lower number, the higher priority. Task with priority 0 disables all the interrupts and priorities 0 to 8 are used by the OS Kernel
TASK_NAME is a string that helps to identify the task. It is also used to get the task ID.
TASK_ATTRIBUTES. You can use more than one attribute for each task on the list. The allowed Task attributes are:

- Auto start — when MQX starts, it creates one instance of the task.
- DSP — MQX saves the DSP co-processor registers as part of the task's context.
- Floating point — MQX saves floating-point registers as part of the task's context.
- Time slice — MQX uses round robin scheduling for the task (the default is FIFO scheduling).

CREATION_PARAMETER is the parameter to be passed to the task when it is created.
TIME_SLICE. Time slice (in milliseconds usually) used for the task when using round-robin scheduling.

# MQX_Template_List

```
{ HELLO_ID, hello_task, 0x1000, 8,
  "hello_task",
   MQX_TIME_SLICE_TASK, 0L, 100},
```

```
{ WORLD_ID, world_task, 0x3000, 9,
  "world_task",
   MQX_AUTO_START_TASK, 0L, 0},
```

```
{ LED_ID, led_task, 0x2000, 10,
   "LED Task",
   MQX_AUTO_START_TASK |
     MQX_TIME_SLICE_TASK, 0L, 50},
```

27

Which tasks auto start?

Answer: world and led

Which task has highest priority?

Answer: hello (lower number has higher priority)

Which task would be the first to run?

Answer: world

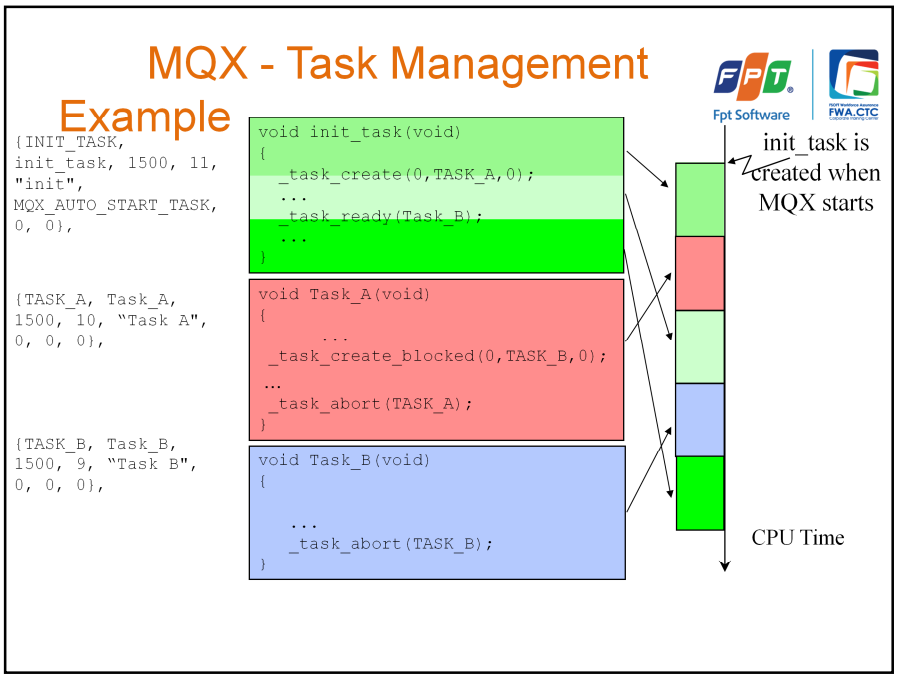
Which task uses Round Robin scheduling?

Answer: hello and world

How long each slice?

Answer: hello is 100ms and led is 50ms

At least one task must be set to MQX_AUTO_START_TASK




Task flow example

Init_task starts first because of the MQX_AUTO_START_TASK configuration flag

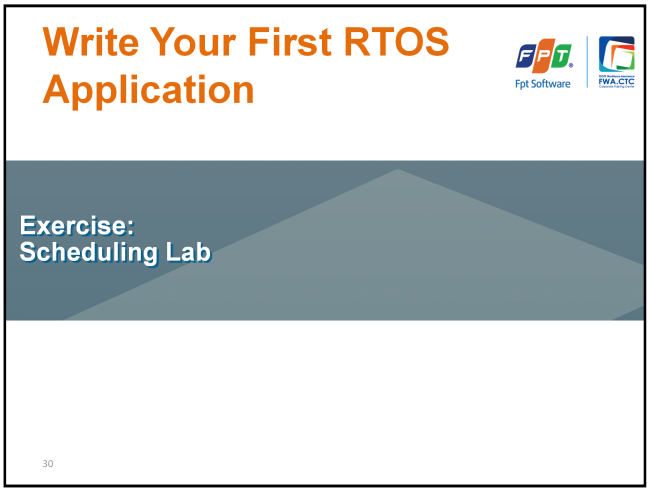
It continues until it creates TaskA. Because Task A has a higher priority, it begins to run.

Inside Task A, it creates Task B, but puts it in a blocked state. Then Task A ends.

Control goes back to the init_task(). It continues until it makes Task B ready. Since Task B is a higher priority, it then begins to run. It goes until the task terminates itself.

Then control goes back once again to the init_task, until it also ends.

# Exercise




**The Exercise includes 2 tasks**

- Main Task:
    - AUTOSTART, priority 9
    - Prints out welcome
    - Loops, incrementing x
- Print Task:
    - AUTOSTART, priority 10
    - Loops, printing the value of x

**Follow the directions in Lab 1 to create, compile, and run the lab using** Keil uVision
**Pause the execution to see the main task running**

**When does the print task run?**

## Hands On: Scheduling

- Add the print task to our application
  - Init Task:
    - AUTOSTART, priority 9
    - Prints out welcome
    - Loops, incrementing x
  - Print Task:
    - AUTOSTART, priority 10
    - Loops, printing the value of x
- Add a _time_delay() to block the init task after incrementing x
- Follow the directions in **Lab 2** to compile, download and run the code

31

31

Without the _time_delay() call, the print_task would never get a chance to run. This is called Task Starvation.

_time_delay() puts init_task into the blocked state for the specified amount of time, allowing lower priority tasks the chance to run. Task starvation is something to always be mindful of when designing your software.

# Task Aware Debugging

- Task Aware Debugging is available to analyze tasks running on MQX
- Look at the ready queues and task state
- This will be discussed more detail in later slide

32

# Table of contents

# Why Synchronization?

- **Synchronization may be used to solve:**
  - Mutual Exclusion
  - Control Flow
  - Data Flow
  - Synchronization Mechanisms include:
  - Semaphores
  - Events
  - Mutexs
  - Message Queues
- **The correct synchronization mechanism depends on the synchronization issue being addressed**

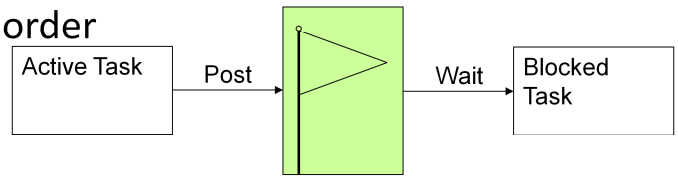




34

## Control Flow with Synchronization

- One task affects another's execution, so that tasks execute in an application-controlled order



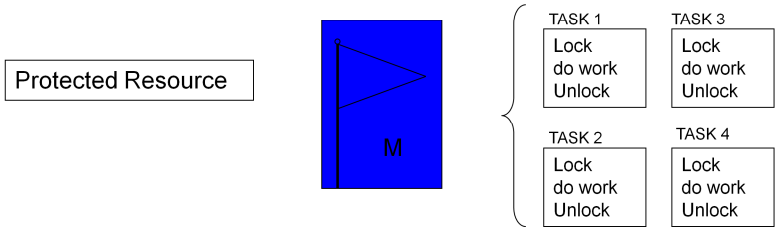

35

35

The active task runs until it has accomplished all it needs to and "post" a key. Posting the key forces the RTOS scheduler to make a "Blocked" task "Active" assuming it has a higher priority the posting task. The task that posted the key will then be placed in a ready fifo queue.

## Mutual Exclusion

- Allowing only one task at a time to access a shared resource
- Resource may be devices, files, memory, drivers, code…
- Mutual exclusion locks the resource




36

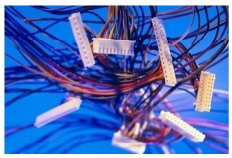
36

A mutex is a special case of a semaphore that only has one key to share.

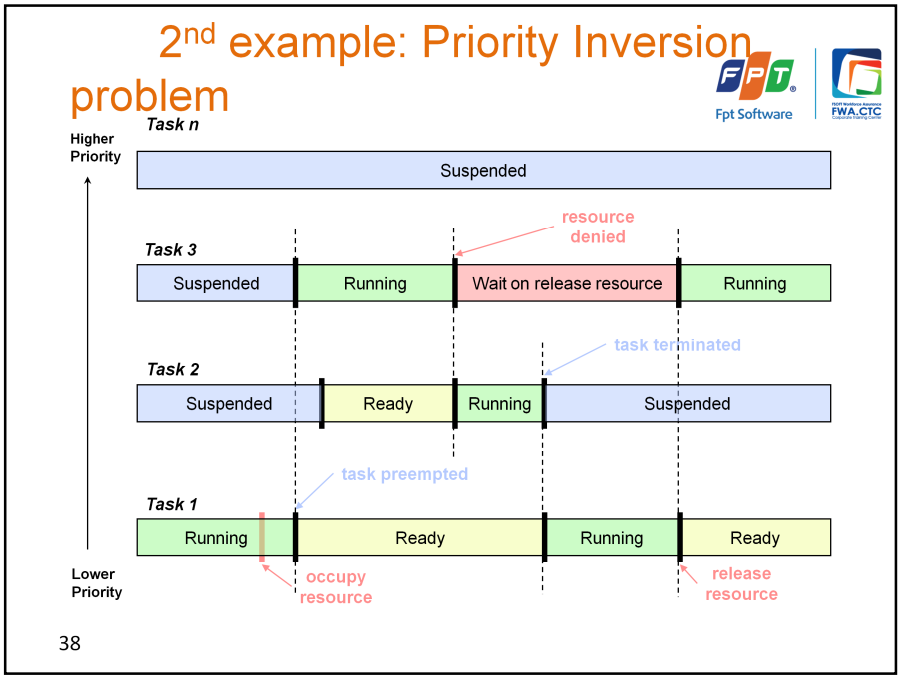
# Competence Condition

- What happens if two tasks access the same resource at the same time?
  - ❑ We call this “competence condition”. When two or more tasks read or write on share a resource at a certain moment

- Why the “competence condition” can be a problem?
  - ❑ Memory corruption
  - ❑ Wrong results
  - ❑ Unstable application
  - ❑ Device conflicts



37




**English**

In the case a high priority task (task3) preempts a low priority task (task1), which occupied a resource, the higher priority task (task3) will get the CPU and run. If the high priority task3 tries to access the same resource, it gets blocked and will wait because the resource was locked by task1. If now a task (task2) with a priority between the two tasks is ready to run, it will get the CPU and execute because the higher priority task is waiting and the lower priority task is preempted. Now, a task with a lower priority block a task with a higher priority. This blocking can be unbounded and then the system can crash.

# Exclusion

- The “competence condition” sound bad… How can I prevent it?

- Don’t worry, we can use exclusion.
  - It is the way to guarantee that if one task is using a shared resource, the other tasks can not use it.
  - You can get the exclusion using operating system’s components created with this purpose. In example:
    - Semaphores
    - Messages queues
    - Mutexes

39

# TAD

- **Agenda**
  - Challenges of debugging MQX applications
  - MQX Task Aware plug-in in classic Keil uVision
    - Installation
    - Screens
    - Compatibility
  - Future Development (IAR)

# Debugging MQX Applications

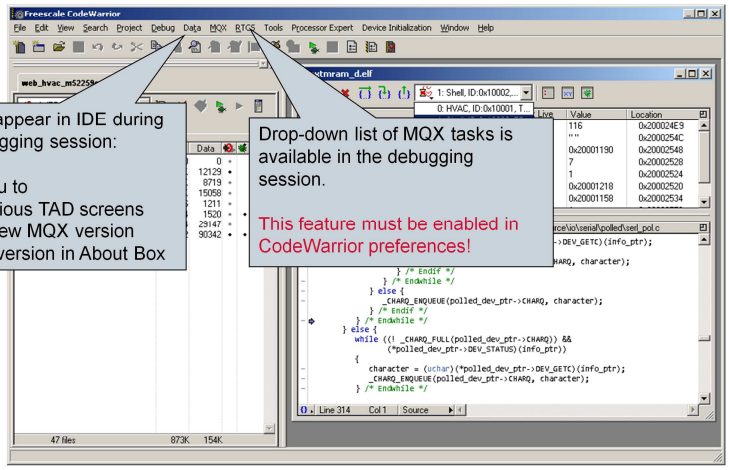
- **Debugging MQX Applications**
  - MQX RTOS and all other MQX components are statically linked to application code.
  - Plain CodeWarrior debugger can be used for basic tasks:
    - Load executable, set breakpoints, execute, break into a running application...
    - Internal MQX data structures can be examined with standard “view variable”
  - Without additional support from TAD, it is a challenge to
    - Understand what tasks are running and where a task-switch occurred
    - Understand values in MQX, RTCS and other data structures (often too complex)

- **What is TAD (Task Aware Debugger)?**
  - Debugger Plug-in helping it to understand task state, descriptors and stack.
  - CodeWarrior IDE Plug-in able to display its content in a easy-to-understand way (selected variables, structure members etc).

41

# What is TAD?

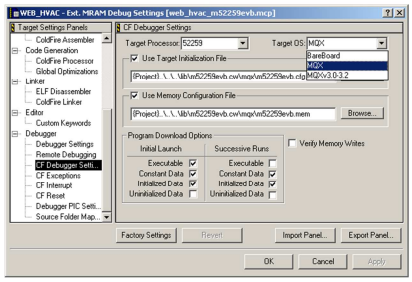
- How does TAD appear in CodeWarrior IDE?



# TAD Installation and Activation

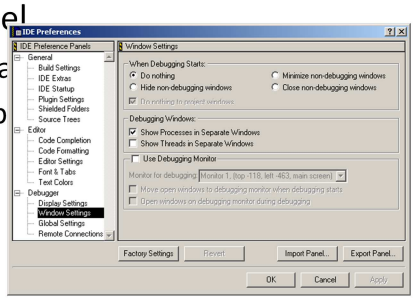
- **Installing TAD**
  - TAD is implemented in **CFrtos_MQX.dll** which must be located in `<CodeWarrior>\bin\Plugins\Debugger\rtos\CFrtos_MQX.dll`
  - Done automatically during MQX InstallShield setup.
  - You need to copy the file manunally (from MQX **tools** folder) if:
    - MQX was not installed using the InstallShield installer (or if it failed to locate CodeWarrior folder)
    - You are migrating to a new CW version (not the CW10!)

- **Activating MQX TAD in Project**
  - Select “MQX” as “Target OS” in *CF Debugger Settings* panel.
  - All projects in MQX distribution are already setup correctly.



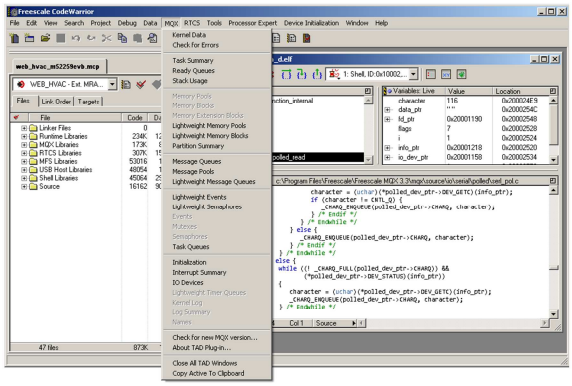
# TAD Installation and Activation

- **Enable Task Drop-down List**
  - This setup must be done (once) manually in the CodeWarrior IDE.
  - Configure CodeWarrior IDE in menu Edit / Preferences.
  - In the *Windows Settings* panel
    - check “Show Processes in Sepa
    - uncheck “Show Threads in Sep
  - This step is described in MQX Release Notes.



# TAD Screens

- **Activate TAD Screens in MQX or RTCS menus**
  - Menu items may be grayed when feature is not compiled in.
- **Key Screens in MQX**
  - Check for Errors
  - Task Summary
  - Stack Usage
  - Memory Blocks
  - Interrupt Summary
- **In RTCS**
  - RTCS Config
  - Socket Summary




45

Stack usage

# Kernel Data

- Version
- Active Task
- In an ISR?
- Scheduler Policy
- Current Time
- Compiler Options

# Other TAD Featuers

- Check for Errors
- Task Summary
- Ready Queues
- Stack Usage
- Memory Usage
- Task Queues
- Initialization
- Interrupt Summary
- IO Devices
- Plus any other features selected at compile time

47

First thing to do if something isn't working is do Check for Errors

Task Summary will tell you what tasks are running and if they have any errors
One common one is "not resource owner" meaning user has tried to manipulate a kernel object it doesn't own (such as trying to free memory twice)
Or sending message twice

Ready queue only shows tasks ready to run and waiting

Stack Usage goes down, tells you how much is being used

Memory blocks tell what things are creating which blocks. Lets you know about memory leaks

# TAD Development

- **Available Today**
  - Classic CodeWarrior Debugger only with Professional Edition
    - CodeWarrior Studio for ColdFire V2-V4
    - CodeWarrior Studio for PowerPC (not public, EAI uses it to support PPC customers)
- **Prototype Avaialble**
  - EAI developed the TAD for IAR Debugger
  - IAR toolset not yet supported by Freescale MQX (question of priorities)
- **Current Development (targeting 2010)**
  - Support Eclipse CodeWarrior 10
  - Java & CORBA wrappers to make use of existing TAD code
  - Resource and prioritization issues (MQX/CW10 support in general)

# When TAD Plug-in is not Available

- **When TAD is not available...**
  - e.g. MQX for ColdFire V1 (Hiwave debugger does not support TAD)

- **TAD "screens" dumped on application console**
  - See `mqx/source/tad` source code compiled in PSP library.
  - `_tad_xxx` dump functions may be called any time by the application.
  - Standard *printf* call is used for the output (works also with redirected stdout, e.g. in telnet sessions).
  - Ideal replacement for host-side TAD on ColdFire V1.
  - Only a few of key TAD screens are implemented today (feel free to contribute).
  - How to use?
    - Add Shell_tad function to the shell commands list when starting Shell().
    - Call _tad_xxx() functions any time in the application:
      - periodically, on event

# Freescale MQX™ Documentation


- MQXUG User Guide
- MQXRM Reference Manual
- MQXUSBHOSTAPIRM USB Host API Reference Manual
- MQXUSBDEVAPI USB Device API Reference
- MQXUSBHOSTUG USB Host User Guide
- MQXRTCSUG RTCS User Guide
- MQXMFSUG File System User Guide
- MQXIOUG I/O Drivers User Guide
- MQXFS Software Solutions Fact Sheets

50

All of these files are in the docs directory in the MQX source code and on freescale.com/mqx

## Further Reading and Training

- Light weight  semaphore
- Messages
- Mutexes
- Events
- Using MQX: RTCS, USB, and MFS
- How to Develop I/O Drivers for MQX

51

All of these documents can be found at freescale.com/mqx and are a sampling of some that you might find the most useful in getting started with  MQX

# Further Reading and Training (Cont.)

- **Videos: www.freescale.com/mqx**
  - ❑ MCF5225x & Freescale MQX introduction
  - ❑ Getting started with MCF5225x and Freescale MQX Lab Demos
  - ❑ And more

- **vFTF technical session videos www.freescale.com/vftf**
  - ❑ Introducing a modular system, Serial-to-Ethernet V1 ColdFire® MCU and Complimentary MQX™ RTOS
  - ❑ Writing First MQX Application
  - ❑ Implementing Ethernet Connectivity with the complimentary Freescale MQX™ RTOS

52

All of these documents can be found at freescale.com/mqx and are a sampling of some that you might find the most useful in getting started with MQX

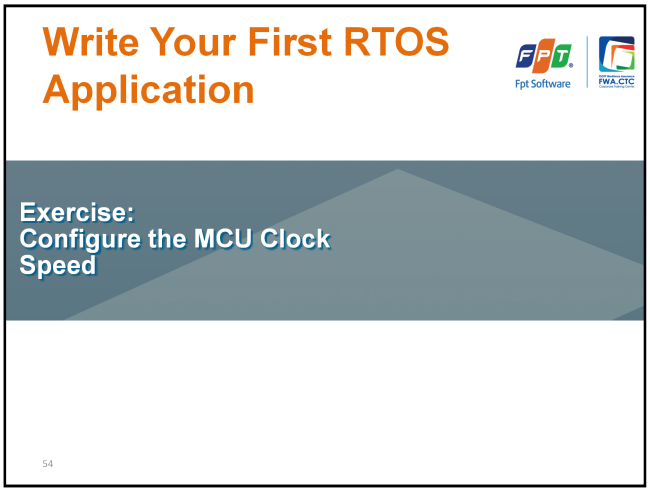
# Further Reading and Training (Cont.)

- **Videos: www.freescale.com/mqx**
  - ❑ MCF5225x & Freescale MQX introduction
  - ❑ Getting started with MCF5225x and Freescale MQX Lab Demos
  - ❑ And more

- **vFTF technical session videos www.freescale.com/vftf**
  - ❑ Introducing a modular system, Serial-to-Ethernet V1 ColdFire® MCU and Complimentary MQX™ RTOS
  - ❑ Writing First MQX Application
  - ❑ Implementing Ethernet Connectivity with the complimentary Freescale MQX™ RTOS

53

All of these documents can be found at freescale.com/mqx and are a sampling of some that you might find the most useful in getting started with MQX




**The Exercise includes 2 tasks**

- Main Task:
    - AUTOSTART, priority 9
    - Prints out welcome
    - Loops, incrementing x
- Print Task:
    - AUTOSTART, priority 10
    - Loops, printing the value of x

**Follow the directions in Lab 1 to create, compile, and run the lab using CodeWarrior 10**

**Pause the execution to see the main task running**

**When does the print task run?**

# Exericise

- Configure the MCU sysclock speed = 60Mhz
- Configure CLKOUT = MCU Sysclock / 2

55

# Summary

By now, you should be able to:

- Understand what an RTOS is and how they can be used
- Create tasks, schedule them and add synchronization using MQX
- Create your own MQX applications

# 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.

58