Hands-On Workshop: MQX™ RTOS and Kinetis Software Development Kit (SDK) Integration

LAT-DES-T1020

Denis Yuji Shimizu

M A R . 2 0 1 5

External Use

Agenda

- Introduction – MQX Basics
- Hello World Example – First MQX for KSDK project
- LwSem Example
- USB Example
- Ethernet Example
- Next events
- Q&A

What is an RTOS?

Real Time Operating Systems

- A real-time operating system manages the time of a microprocessor or microcontroller.
  - **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

Why use an RTOS?

- 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
- Hardware abstraction through device drivers
- Portability of application code to other CPUs
- Efficient interrupt handling
- Resource handling
- Support for upper layer protocols such as:
  - TCP/IP, USB, Flash Systems, Web Servers,
  - CAN protocols, Embedded GUI, SSL, SNMP

Building your first MQX RTOS application

* For IDE specific instructions, see the doc located on the folder: C:\Freescale\KSDK_1.1.0\rtos\mqx\doc\tools

1. Open the workspace file with all MQX RTOS libraries located in the <MQX_DIR>/build/<tool>/<board>/build_libs.*

2. Build the required targets (i.e. Debug) in all projects contained in the workspace.

3. Open the workspace file with all MQX RTOS examples located in the <MQX_DIR>/build/<tool>/workspace_twrk64f120m/build_apps.*

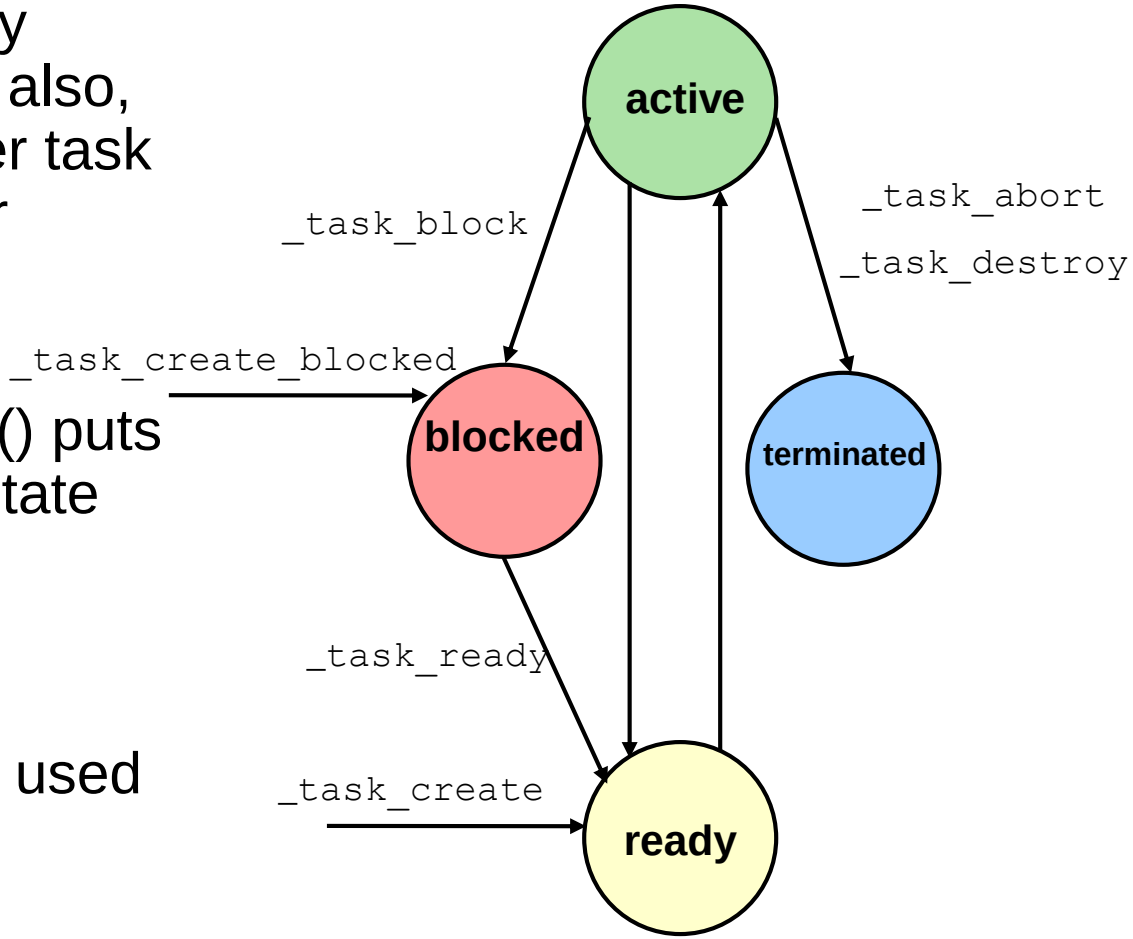
4. Build required targets (i.e. Debug) for a desired application (i.e. Hello_World) project.

5. Run the application.

Tasks

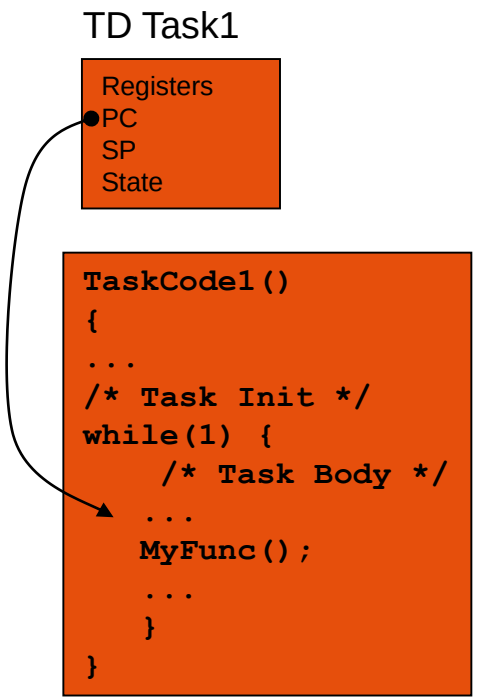
MQX Tasks

- 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 new task in the ready state and the scheduler runs the highest priority task
- If _task_create_blocked is used the task is not ready until _task_ready() is called

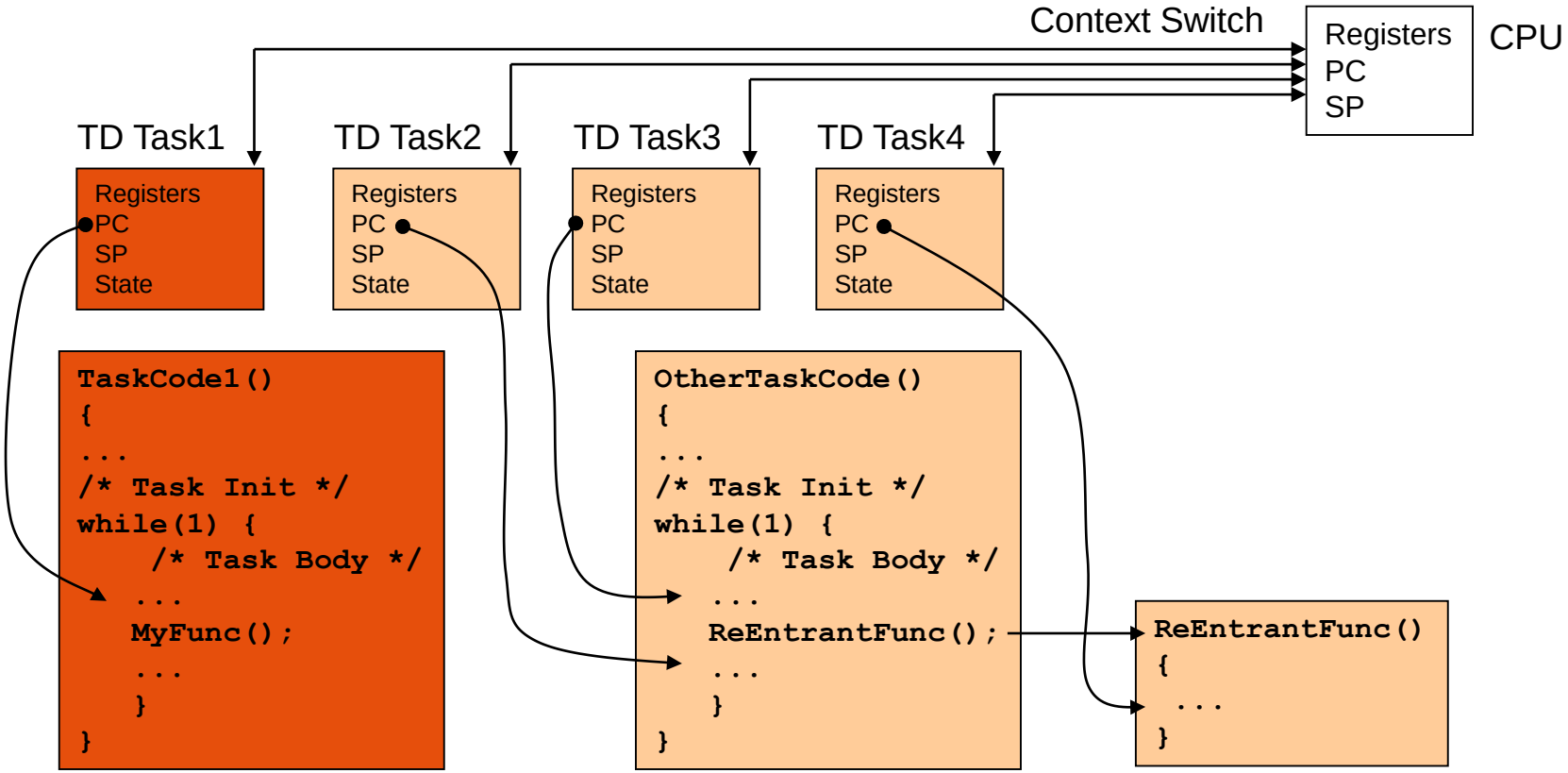


Overview of Task Management

- A task is a unique instance of a task template
- A task is defined by its task descriptor, which includes:
  - context
    - program counter
    - stack
    - registers
  - priority
  - resources
  - task ID
  - task-specific parameters



Overview of Task Environment



Task attributes

- Define the type of task that is created; can be any combination of:
  - MQX_AUTO_START_TASK
  - MQX_DSP_TASK
  - MQX_FLOATING_POINT_TASK
  - MQX_TIME_SLICE_TASK (roundrobin)
  - Or 0 for none of the above
- If MQX_TIME_SLICE_TASK isn’t specified, the task's scheduling policy is FIFO

Creating tasks dynamically

- To create a task dynamically and make it ready:
  - `_task_id _task_create(`
    processor_number`,`
    template_index`,`
    parameter_to_new_task`)`
  - if *template_index* = 0, *parameter_to_new_task* is a pointer to a task template
- The task descriptor and stack are allocated and initialized
- The task-scheduling policy is obtained from the task template; one of FIFO (default) or roundrobin (timeslice)
- The task is put in its ready queue

continued

Creating tasks (con't)

- To create a task dynamically, but not make it ready:

  `_task_id` **`_task_create_blocked`**`(`
  processor_number`,`
  template_index`,`
  parameter_to_new_task`)`

  – if *template_index* = 0, *parameter_to_new_task* is a pointer to a task template

- To begin executing the blocked task

  `_task_ready``(`*task_id*`)`

Restarting tasks

- To restart a task:

```
_mqx_uint  _task_restart(
  task_id,
  param_to_restarted_task,
  blocked)
```

  - closes all queues the task has open
  - releases all the task's resources
  - frees all memory associated with the task's resources
  - restarts the task with the same task descriptor, task ID, and task stack
  - if *blocked* = TRUE, restarts the task in a blocked state

Terminating tasks

- Releases all of a task's MQX-managed resources:
  - dynamically allocated memory
  - message queues
  - messages
  - mutexes
  - semaphores
  - lightweight semaphores
- To disallow the termination of tasks (to save code space):
  - set the compile-time configuration option, MQX_TASK_DESTRUCTION, to 0 (the default is 1)

Terminating tasks (cont'd)

- To terminate a task immediately :

  _task_destroy(*task_id*)

- To terminate a task and run an exit handler to do any application-specific clean up:

  _ **_task_abort**(*task_id*)

  The task executes its exit handler and then calls **_task_destroy**()

  To set the task exit handler:

  _task_set_exit_handler(
  task_id,
  exit_handler_function_address)

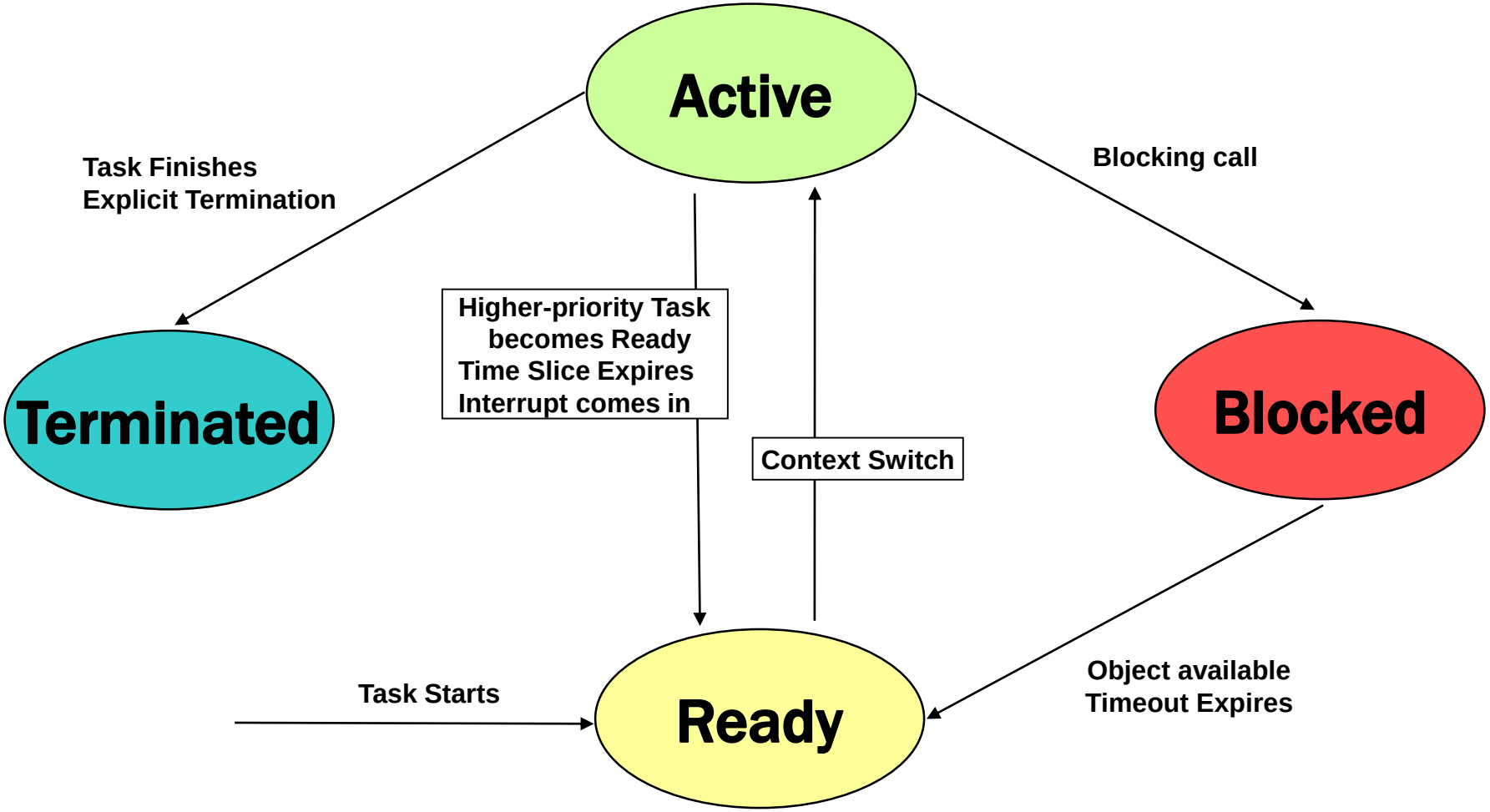
- If a task just returns, MQX will perform the equivalent of **_task_abort()**

Scheduling

Scheduler Overview

- A system consists of multiple tasks
- Tasks take turn running
- Only one task is active (has the processor) at any given time
- MQX manages how the tasks share the processor (context switching) via the Scheduler

Task States

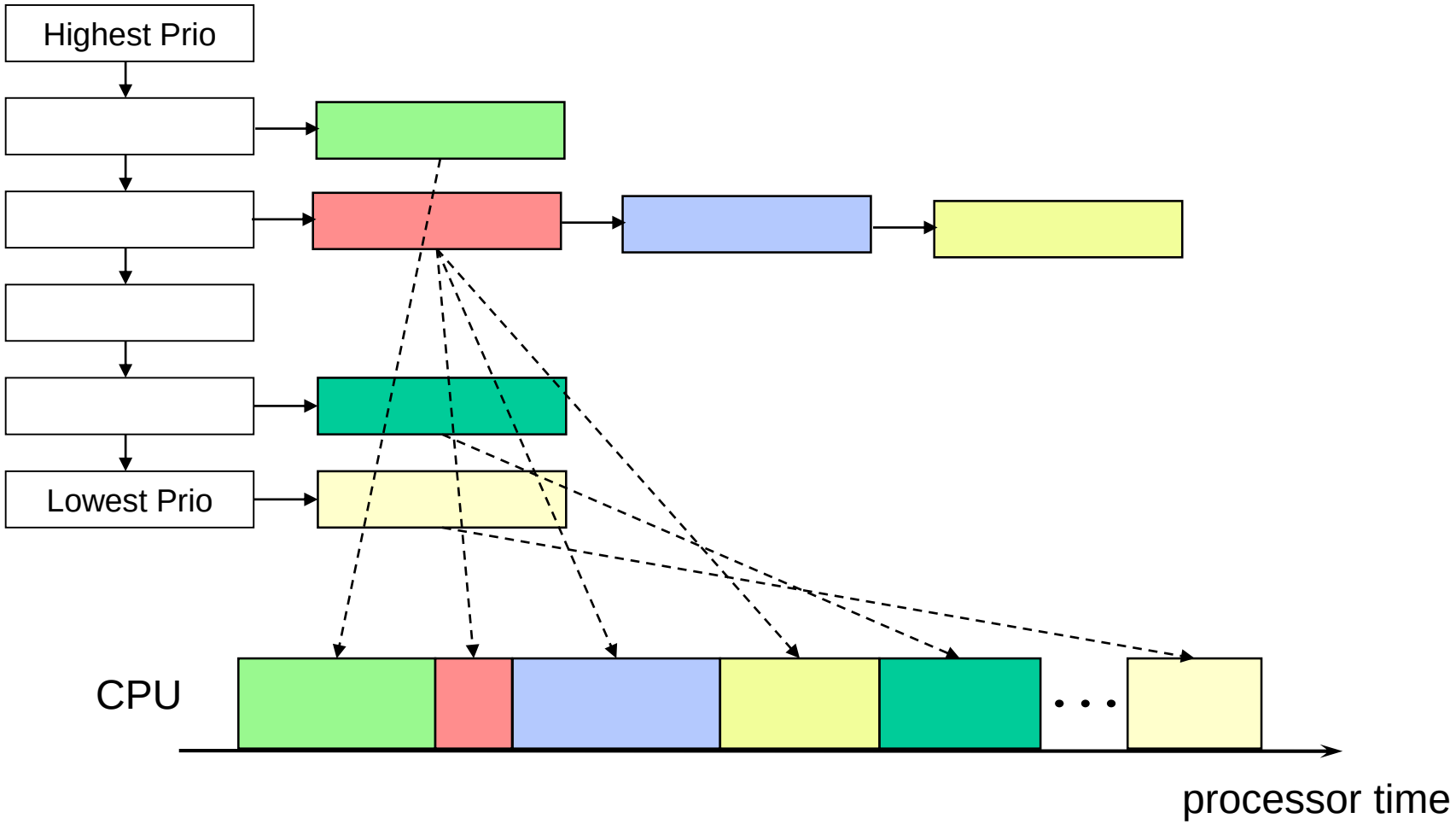

Active
Task Finishes
Explicit Termination
Blocking call
Higher-priority Task
becomes Ready
Time Slice Expires
Interrupt comes in
Terminated
Context Switch
Blocked
Task Starts
Ready
Object available
Timeout Expires

MQX Scheduler

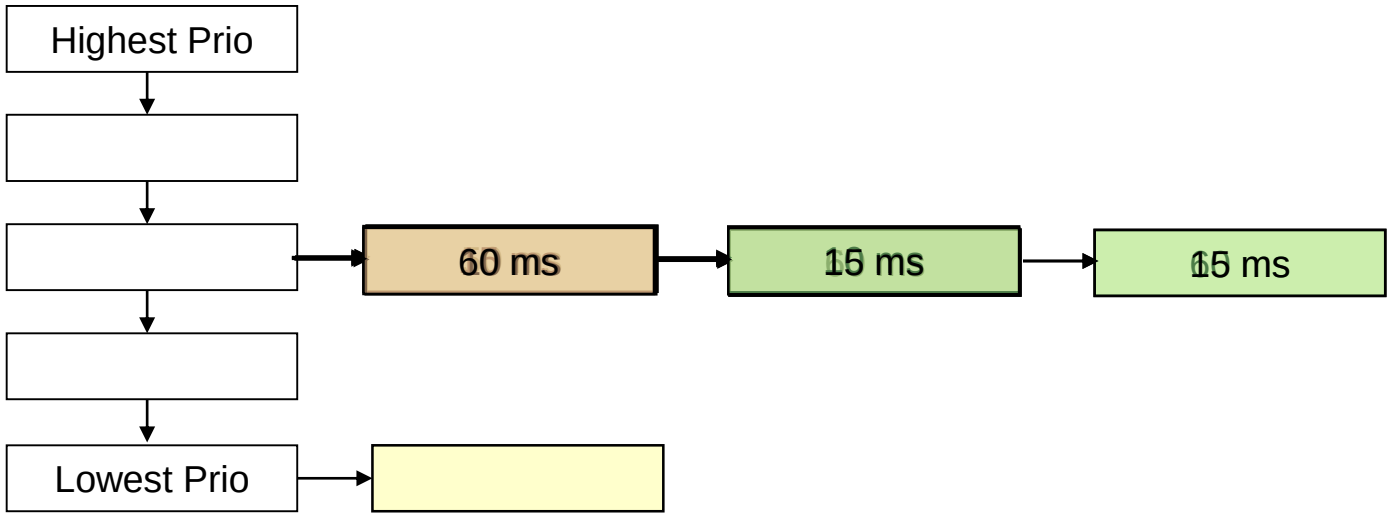
- MQX Scheduler Supports These Policies:
  - FIFO (also called priority-based preemptive)
    - The active task is the highest-priority task that has been ready the longest.
  - Round Robin
    - The active task is the highest-priority task that has been ready the longest without consuming its time slice
  - Explicit Scheduling (optional)
    - Task queues are used to explicitly schedule tasks or to create more complex synchronization mechanisms*

* An application can specify a FIFO or round robin scheduling policy when it creates the task queue.

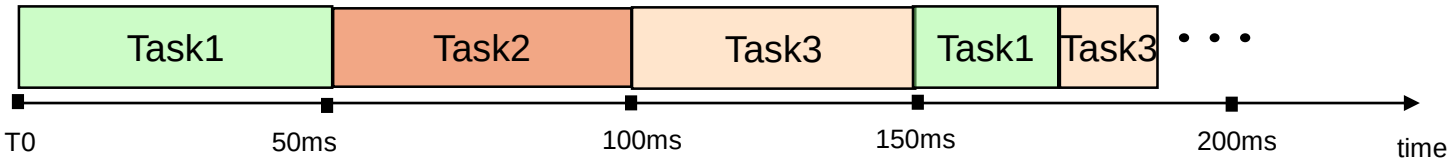
Priority Based FIFO Scheduling



Round-Robin Scheduling



Time Slice = 50ms



MQX - Task management example

```
{INIT_TASK,
init_task, 1500, 11,
"init",
MQX_AUTO_START_TASK,
0, 0},
```

```
void init_task(void)
{
  _task_create(0,TASK_A,0);
  ...
  _task_ready(Task_B);
  ...
}
```

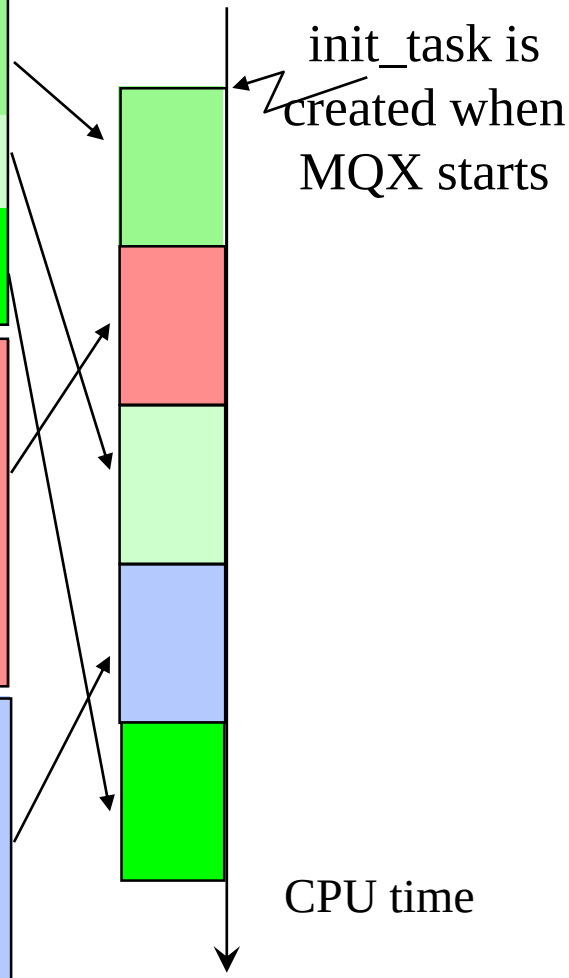
```
{TASK_A, Task_A,
1500, 10, "Task A",
0, 0, 0},
```

```
void Task_A(void)
{
      ...
 _task_create_blocked(0,TASK_B,0);
 ...
 _task_abort(TASK_A);
}
```

```
{TASK_B, Task_B,
1500, 9, "Task B",
0, 0, 0},
```

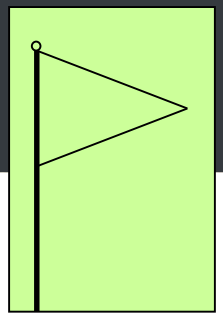
```
void Task_B(void)
{

   ...
   _task_abort(TASK_B);
}
```



Creating first MQX for KSDK example

Semaphores



Semaphores

- A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:
  - control access to a shared resource (mutual exclusion)
  - signal the occurrence of an event
  - allow two tasks to synchronize their activities

- A semaphore is a “token” that your code acquires in order to continue execution.

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

How semaphores work

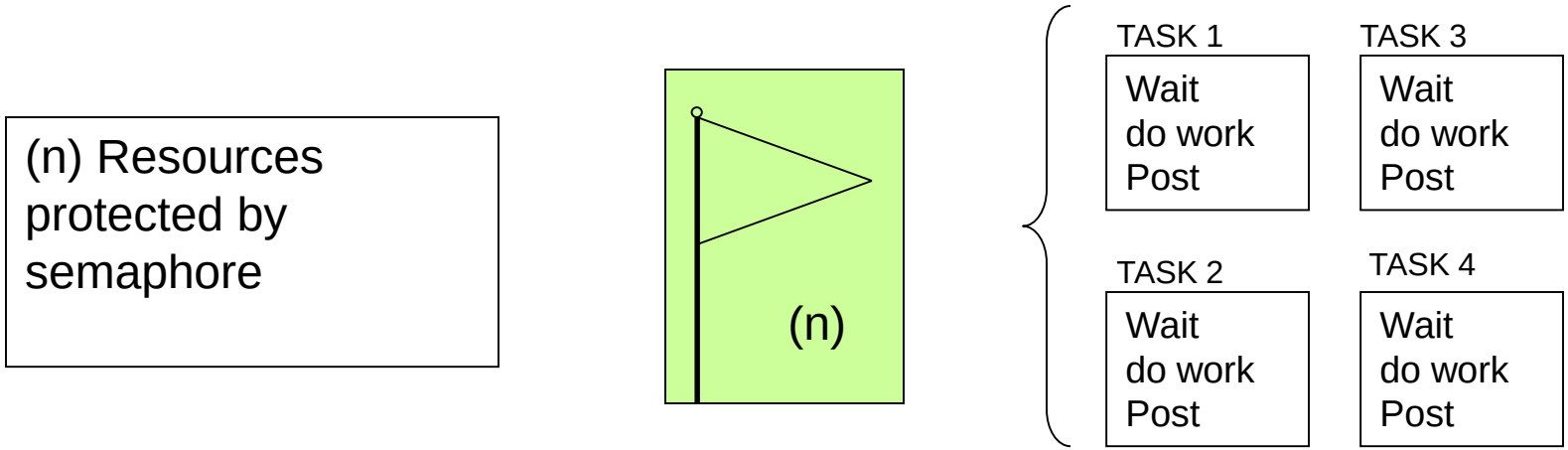
- A semaphore has:
  - counter — maximum number of concurrent accesses
  - queue — for tasks that wait for access

- 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

How Semaphores Work



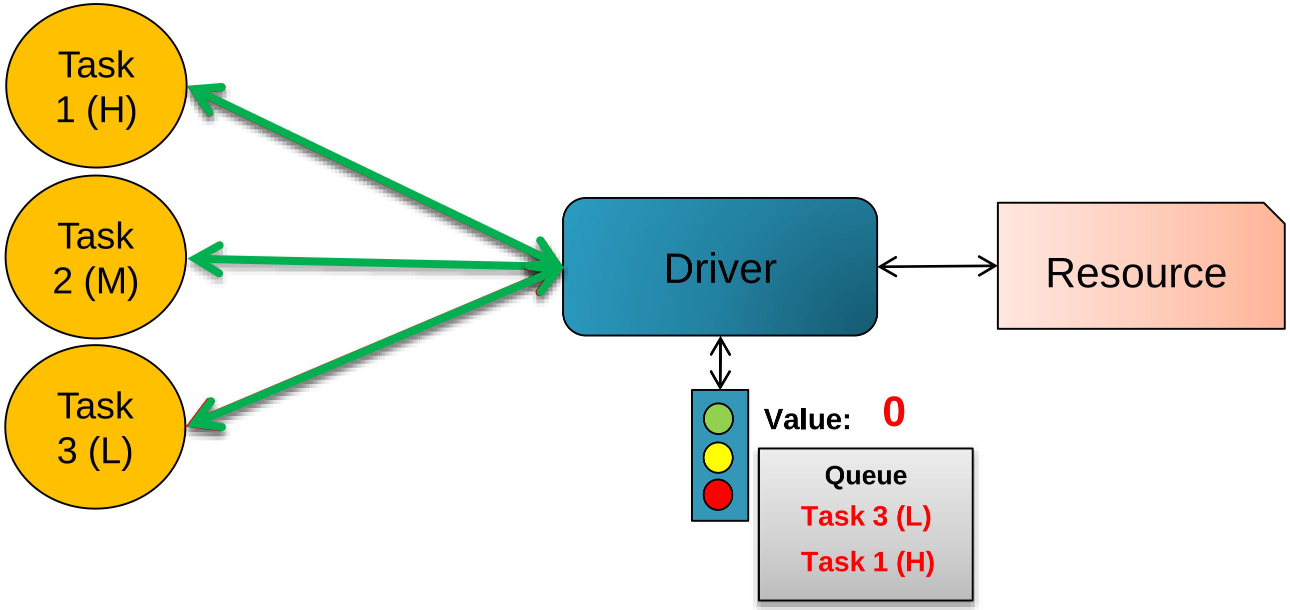
- A semaphore has:
  - Counter – maximum number of concurrent accesses (n)
  - Queue – for tasks that wait for access
- If a task waits for a semaphore
  - If n>0, n is decremented and task runs
  - If n is 0, task is put on the waiting queue
- When a task releases (posts to) a semaphore
  - If a task is in the queue, task is readied per the queuing policy
  - No queued tasks, n is incremented

Strictness

- 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 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 example

- INITIALIZE (also called *CREATE*)
- WAIT (also called *PEND*)
- *SIGNAL (also called POST)*
- The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty.

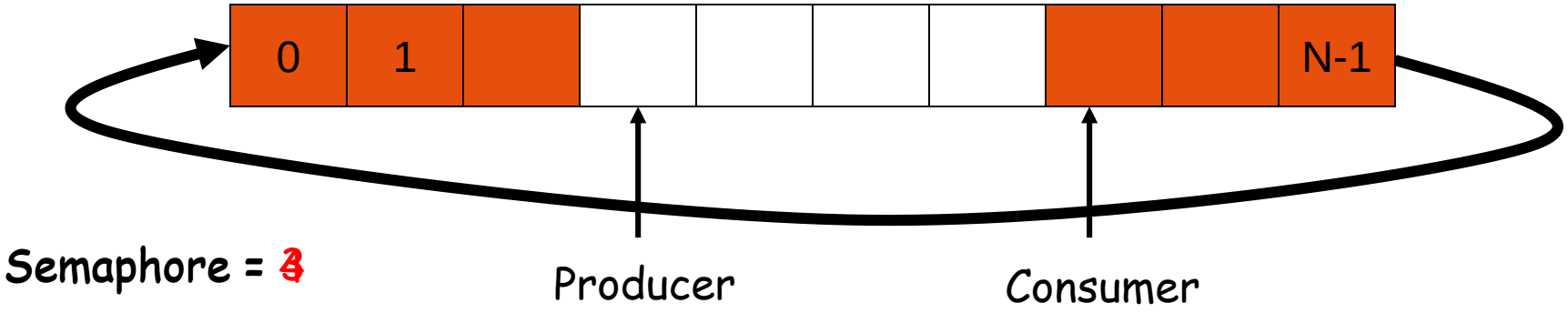


Task 3 releases the resource. Because queue is empty value is incremented

e. the waiting list.

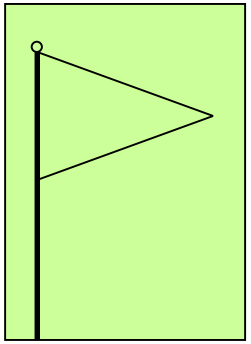
Counting semaphores example

- We have a Bounded buffer: size ‘N’
  - The access entry is 0… N-1, then “wrap around” to 0 again
- Producer Task writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer Task reads data from buffer
  - Should not try to consume if there is no data
- A semaphore is used to have the control of the available space in the buffer.



Lightweight Semaphore Overview

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



Create
Wait
Post
Poll
Wait for
Wait ticks
Wait until
Destroy

Using lightweight semaphores

- To declare a lightweight semaphore:

```
LWSEM_STRUCT sem;
```

- To create a lightweight semaphore:

```
_mqx_uint _lwsem_create(lwsem_ptr,initial_count)
```

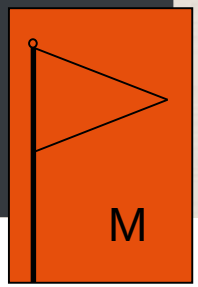
- To wait for a lightweight semaphore

```
_mqx_uint _lwsem_wait(lwsem_ptr)
```

- To post a lightweight semaphore:

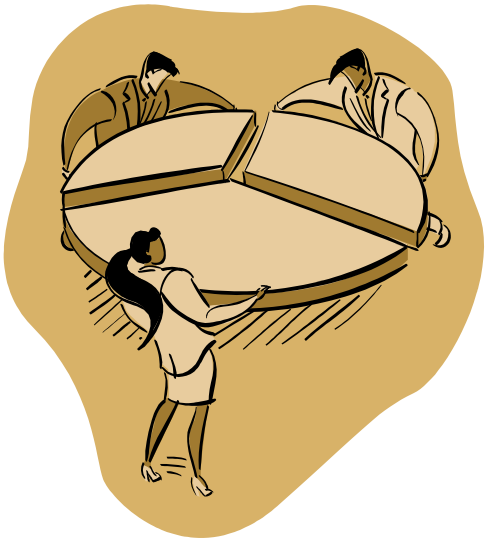
```
_mqx_uint _lwsem_post(lwsem_ptr)
```

Mutex



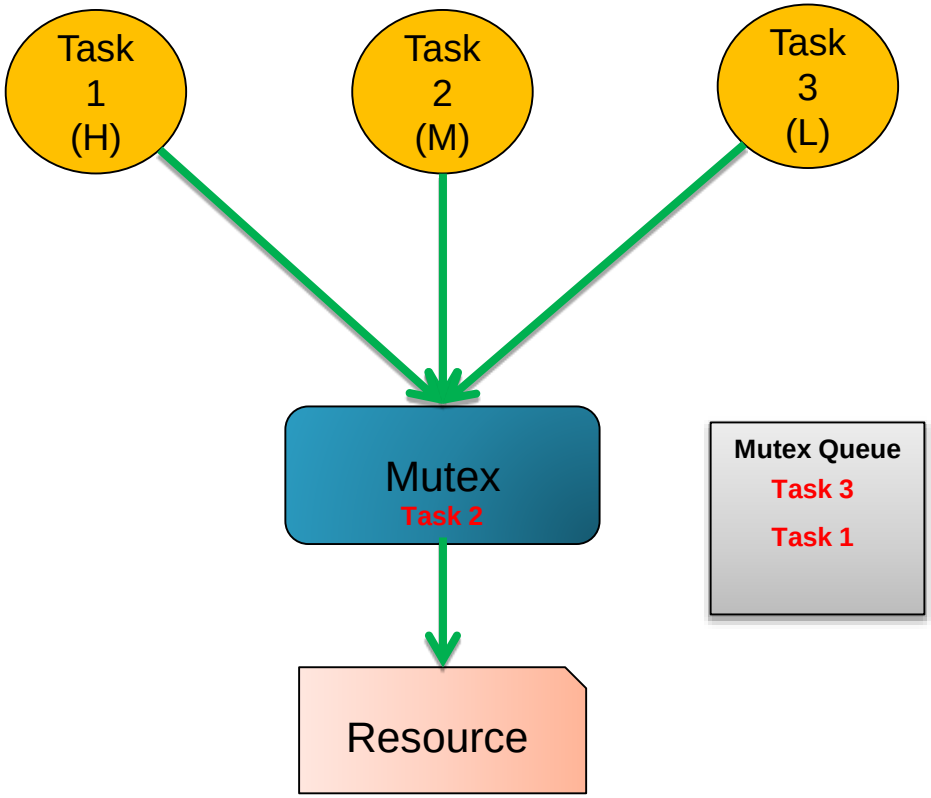
Mutex

- 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.



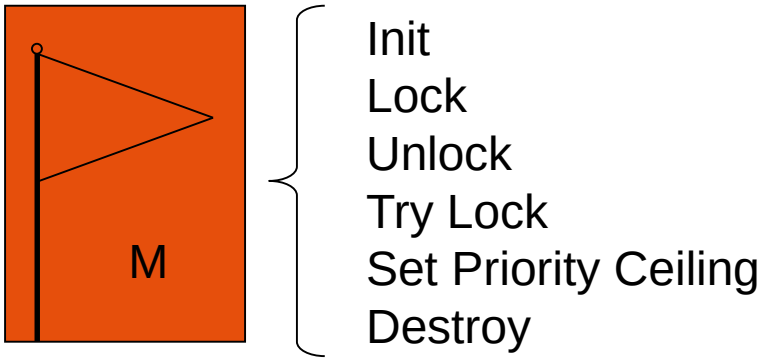
Mutex example

- The task 2 unlocks the Mutex and the task 1 locks the Mutex.
- The task 3 unlocks the Mutex and the task 2 locks the Mutex.
- There are 3 tasks, Task 2 locks the Mutex.
- Task 1 waits in the Mutex queue.
- Task 3 waits in the Mutex queue.



Overview of MQX Mutexes

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



Using mutexes

- To declare a MUTEX:

```
MUTEX_STRUCT mutex;
```

- To initialize the mutex with default attributes:

```
result = _mutex_init(mutex_ptr, NULL)
```

- To lock a mutex:

```
_mqx_uint _mutex_lock(mutex_ptr)
```

- To unlock a mutex:

```
_mqx_uint _mutex_unlock(mutex_ptr)
```

Mutual Exclusion Hands On

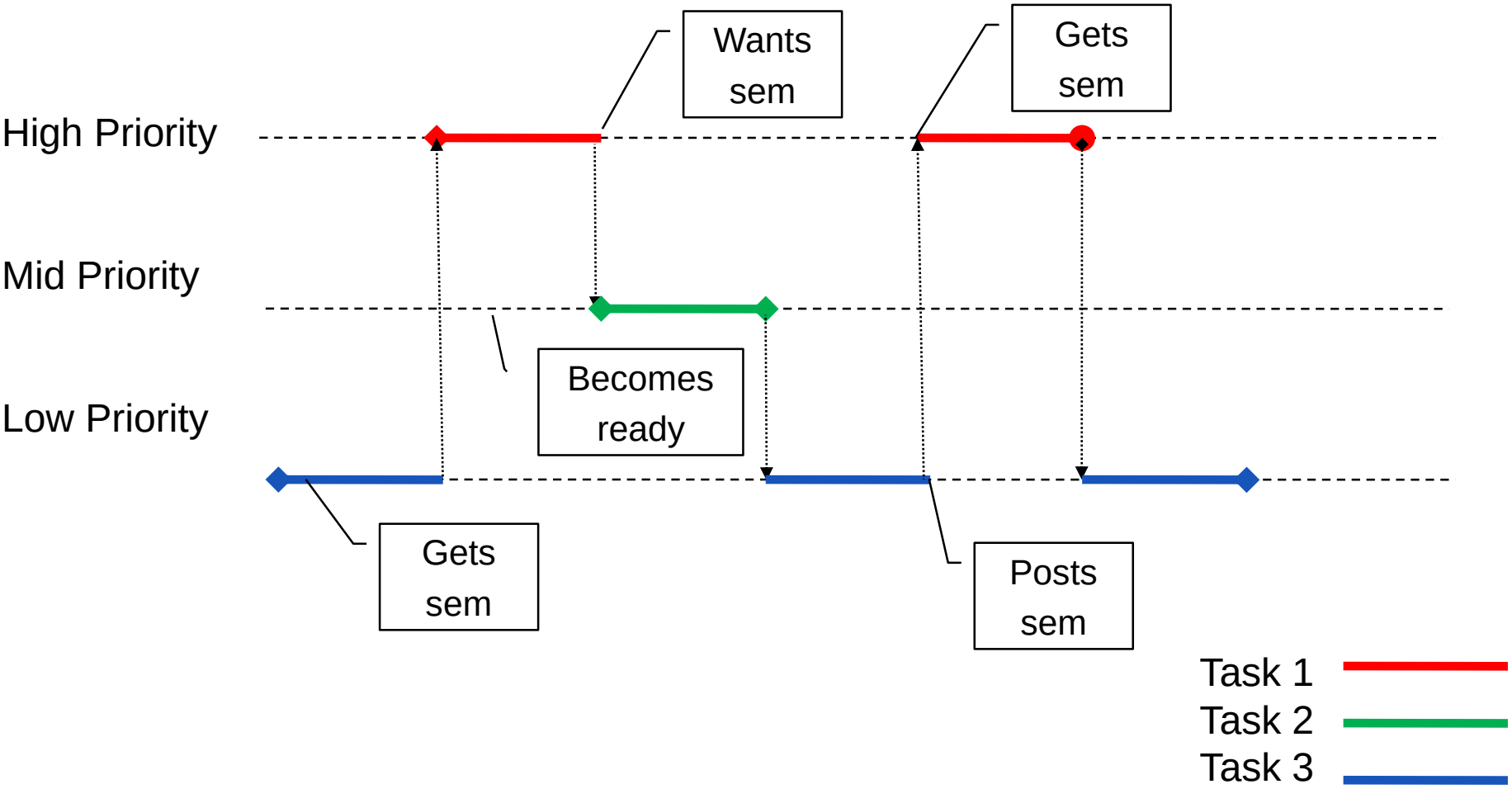
- Replace the lightweight semaphore calls with equivalent mutex calls
- Because mutexes are optional, you will need to include mutext.h
- Run
- Use TAD to view the mutex

Priority Inversion

Priority inversion

- Occurs when a low-priority task causes a higher-priority task to block
- In the following example:
  - Task_1 is priority 1 (highest)
  - Task_2 is priority 2
  - Task_3 is priority 3 (lowest)
  - Task_2 prevents Task_1 from running

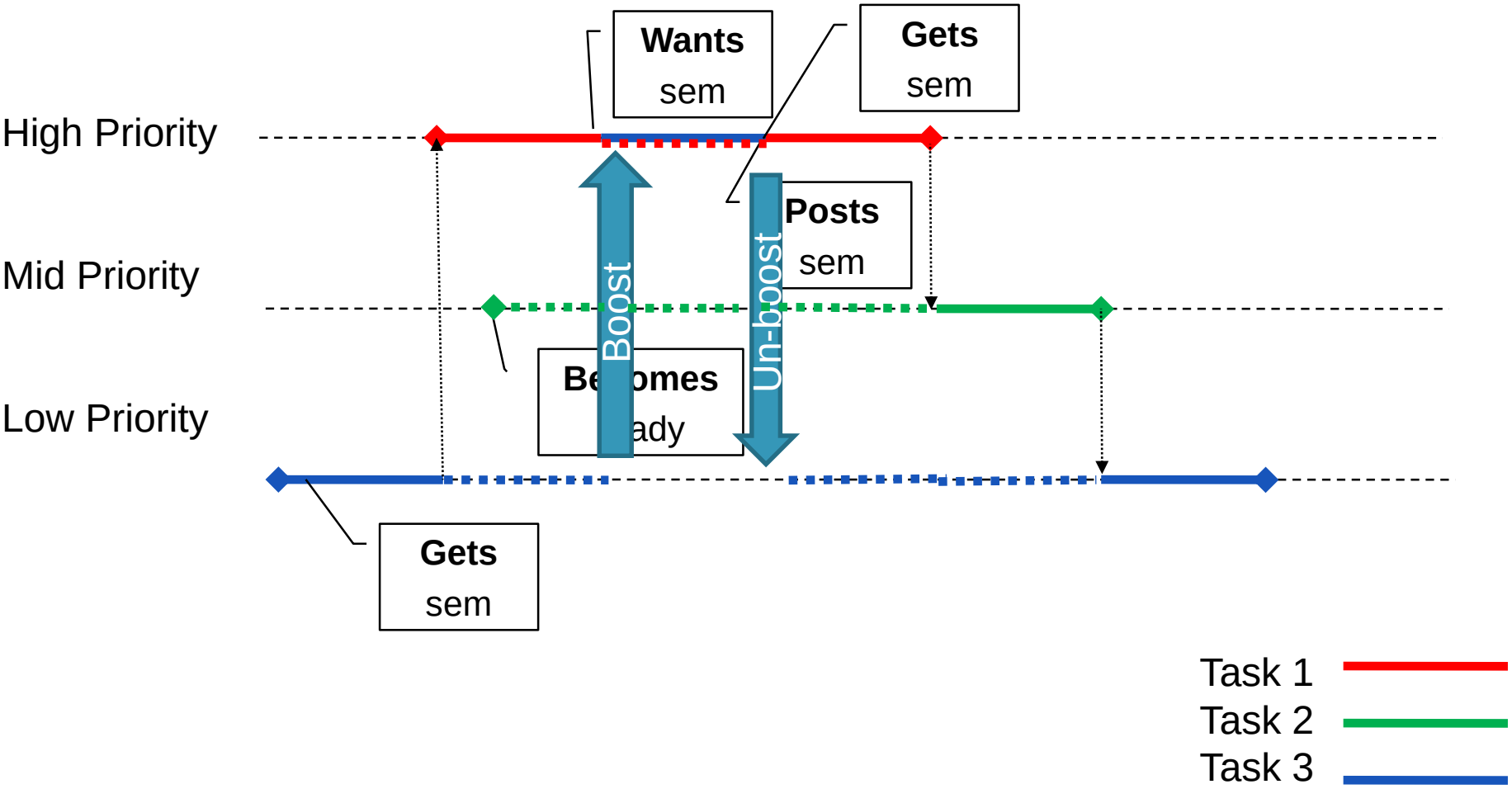
Priority inversion illustrated



Priority Inheritance

- With priority inheritance, if a task is waiting for a semaphore or mutex and has a higher priority than the task that has the semaphore or mutex, the priority of the task that has the semaphore or mutex is temporarily raised to the priority of the waiting task
- The lower-priority task finishes its work and releases the semaphore or mutex
- This allows the higher-priority task to run

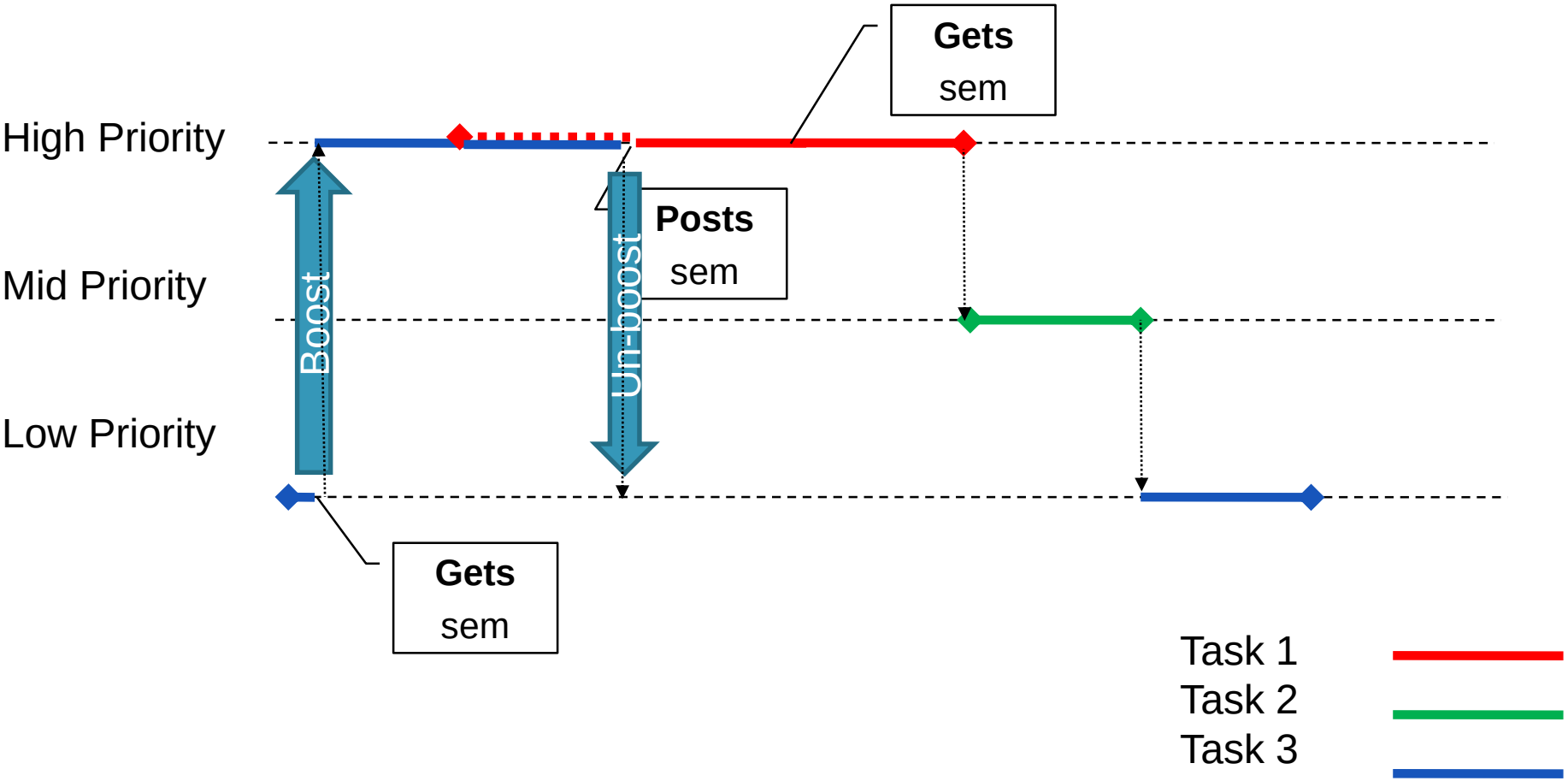
Priority inheritance illustrated



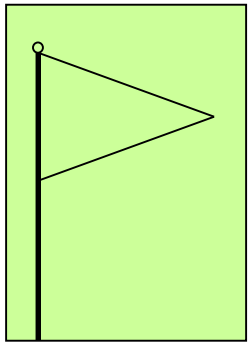
Priority Protection

- When a task tries to lock a mutex that has priority protection, and it isn't at least the same priority the mutex specifies, the task's priority is increased to that of the mutex until the task unlocks the mutex

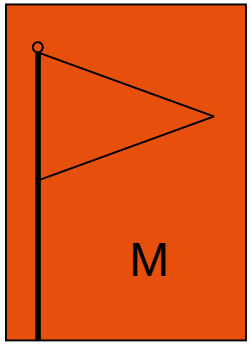
Priority inheritance illustrated



Binary semaphore vs. Mutex



- Mutex
  - strict
  - binary
  - Ownership
  - priority queuing
  - Priority protection
  - used for mutual exclusion only
  - No timeout



- Semaphore
  - Strict or non-strict
  - Binary or counting
  - priority queuing
  - used for event notification and mutual exclusion
  - Timeout

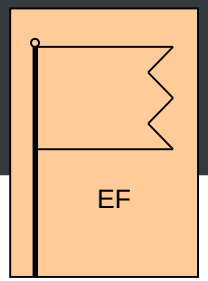
Synchronization

- Control Flow

Control Flow

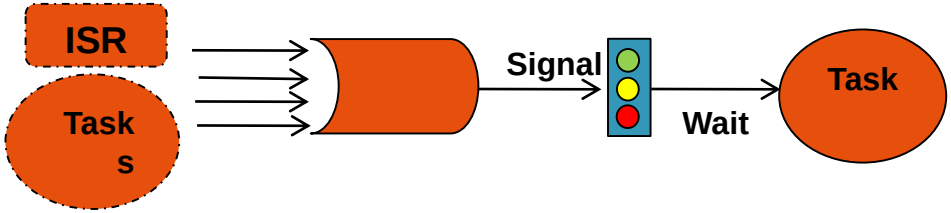
- Control Flow provides a mechanism for a task or ISR to resume execution of one or more other tasks
- Mutual Exclusion is used to prevent another task from running
- Control Flow is used to allow another task to run
- Not all synchronization objects are suitable for control flow:
  - Good: Non-strict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

Events

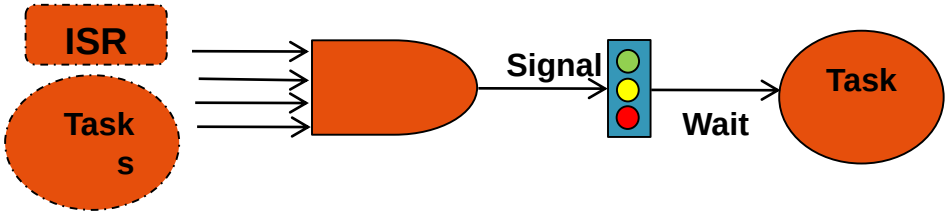


Event Flags

- Event flags are used when a task needs to synchronize with the occurrence of multiple events. The task can be synchronized when any of the events have occurred. This is called disjunctive synchronization (logical OR).



- A task can also be synchronized when all events have occurred. This is called conjunctive synchronization (logical AND).

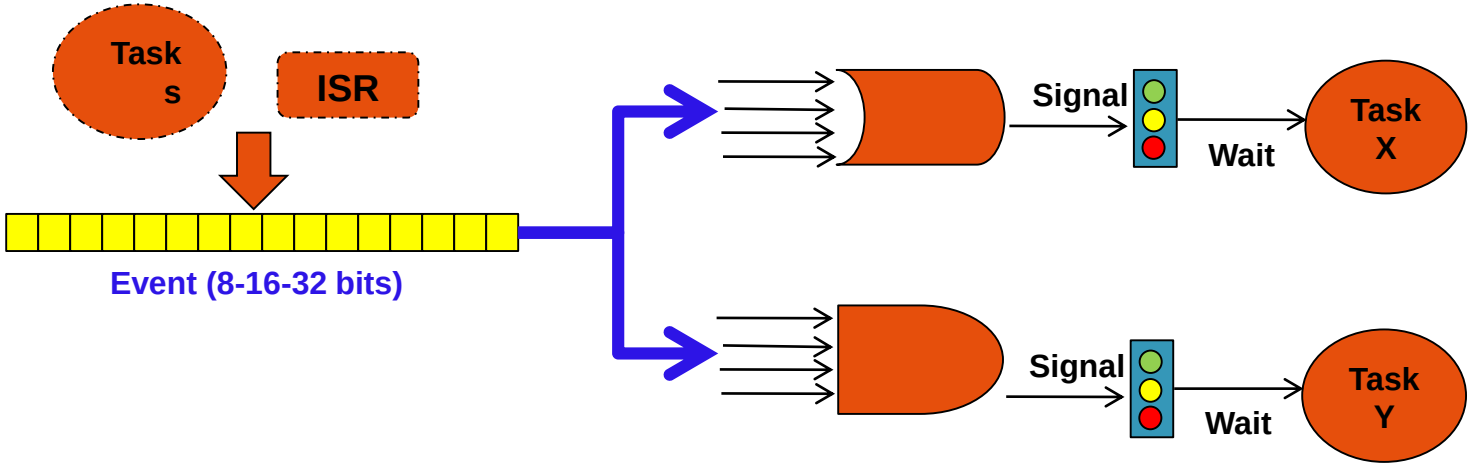


- Kernels supporting event flags offer services to SET event flags, CLEAR event flags, and WAIT for event flags (conjunctively or disjunctively).

Event Flags

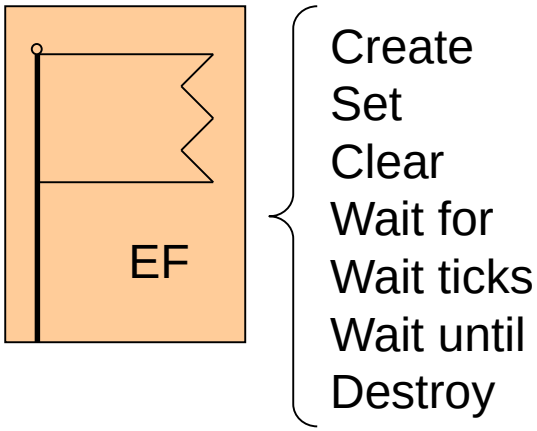
- Tasks and ISRs can set or clear any event in a group.

- A task is resumed when all the events it requires are satisfied. The evaluation of which task will be resumed is performed when a new set of events occurs.

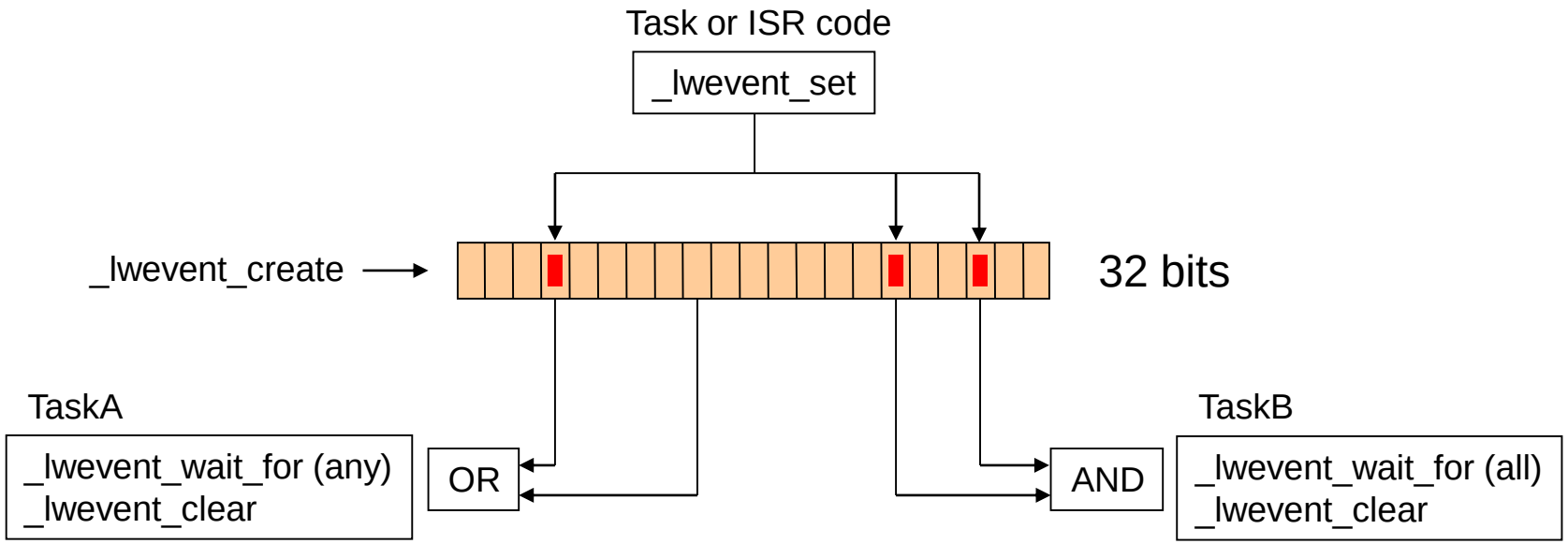


Lightweight Events Overview

- MQX Optional Component
- Similar to regular Event Groups, except:
  - Identified by address
  - Connections are not needed



Lightweight Event Flag Group



- Tasks pend on either a combination of all bits (logical and) or any of the bits (logical or)
- Unless configured for auto-clear, application must explicitly clear bits

Lightweight event groups

- To declare a lightweight event group:
  `LWEVENT_STRUCT lwevent;`

- To create a lightweight event group:
  `_mqx_uint _lwevent_create(`*lwevent_ptr, flags*`)`
  - *flags* — NULL or LWEVENT_AUTO_CLEAR

- To set the specified event bit or bits in a lightweight event group:
  `_mqx_uint _lwevent_set(`*lwevent_ptr, mask*`)`
  - if this is an auto-clearing event group, only one task is made ready

- To wait for a number of ticks (in ticks):
  `_mqx_uint _lwevent_wait_ticks(`*lwevent_ptr,mask,all,ticks*`)`
  - If *ticks* = 0, the wait is unlimited

- To clear the specified event bit or bits in a lightweight event group:
  `_mqx_uint _lwevent_clear(`*lwevent_ptr, mask*`)`

Events vs. Semaphores for Control Flow

- Events
  - One-to-many
  - Multiple Event bits
  - AND / OR conditions
  - Auto-clear option
- Semaphores
  - One-to-one
  - Single condition
  - Priority inheritance
  - Queues successive signals

Next Events

Próximos treinamentos

Treinamento DwF:

Tecnologias Multicore para IoT e Soluções de Software MQX

- **Multicore**
  - Palestra: Apresentando os processadores QorIQ LS1 para aplicações em IoT, M2M e Redes, e a plataforma Yocto Project™
  - Hands-On**:** Introdução ao desenvolvimento de sistemas embarcados com abordagem multicore assimétrica (Toradex)
- **Ferramentas de Software para o Desenvolvimento**
  - Hands-On: Integração do RTOS MQX e o Kinetis Software Development Kit (SDK)
  - Palestra: Acelerando o desenvolvimento de firmware do Kinetis utilizando os novos drivers de periféricos, stacks e middleware do SDK

Data: 19 de Março de 2015

Horário: 09h00 - 17h00

Locação: SENGE-RS,

Investimento: Gratuito
**As Vagas são limitadas.*

Inscrições: *goo.gl/y7zfE2*

Webinar:

Go Multicore: Facilitando o desenvolvimento com os processadores QorIQ LS1 com arquitetura Layerscape e núcleo ARM®

- A família de processadores QorIQ LS1 está baseada em uma arquitetura Layerscape
- Entrega uma alta performance de software com baixo consumo energético, dissipando menos de 3 Watts sob performance de 6000 CoreMark®
- Diversas aplicações industriais (ex.: IoT, M2M, PLC, Profibus, Fieldbus, Gateway) e de Redes (ex.: Roteador, WLAN, SDN)
- Ferramentas de software e hardware para o desenvolvimento

Data: 14 de Abril de 2015

Horário: 14h00 - 15h00

Locação: Online

Inscrições: *goo.gl/S2yU8V*

FTF 10th ANNIVERSARY FREESCALE TECHNOLOGY FORUM 2015

Secure Embedded Processing Solutions for the Internet of Tomorrow

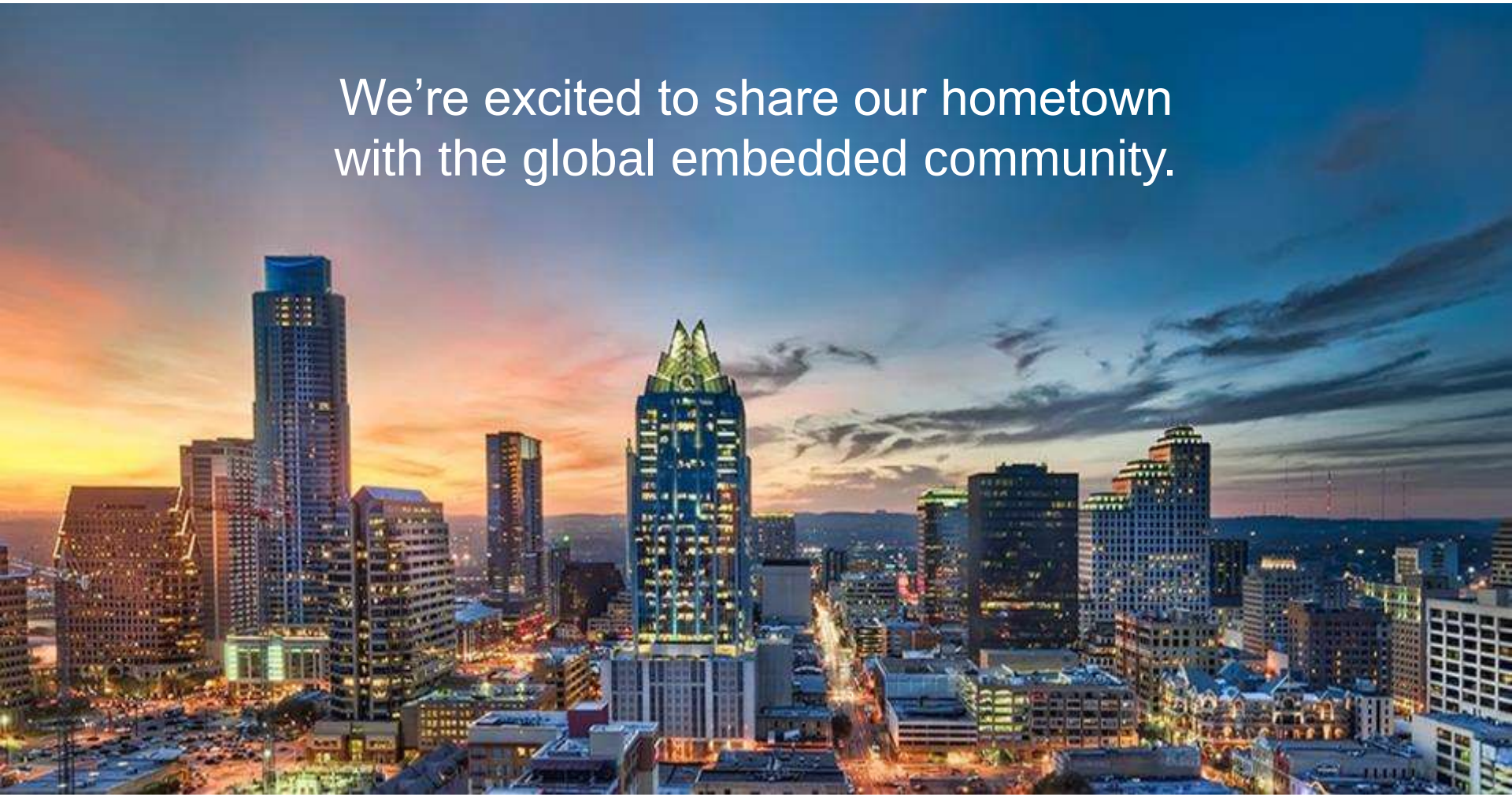
June 22-25, 2015

JW Marriott Austin

FTF.freescale.com

freescale™

FTF Moves to Austin



From live music, to food trucks, to bats, Austin has a lot to offer, so stay a few extra nights and enjoy.

Live Music >

Food & Drinks >

Attractions >

JW Marriott Austin

- This year's FTF will be held at the JW Marriott Austin. This new luxury hotel in downtown Austin will bring FTF to a whole new level.
  - 1,012 luxurious guest rooms and suites
  - Within walking distance of the Capitol
    and exciting attractions
  - 15-minute drive from Austin International Airport
  - Opened February 2015
  - Discounted room rate of $217/night, plus tax. Freescale has also secured discounted rates at a number of surrounding hotels. Special rates are only available during the online FTF registration process.

jwmarriottaustin.com

Technology Lab

Whether you're looking for your next big idea, help with your design, or the ability to expand your network, the technology lab has the answers you need.

Attendee	Extra Early Bird March 16 – May 3	Early Bird May 4 – May 31	Regular June 1 – June 22
Regular	$850	$1,050	$1,300
Alumni	$650	$850	$1,100

All fees are specified in U.S. dollars Cost includes access to all technical sessions and general sessions, the technology lab, event party and meals. FTF alumni receive $200 off current price.

Visit FTF.freescale.com/register for more information

FTF

FREESCALE TECHNOLOGY FORUM 2015

BRAZIL

September 15, 2015

Grand Hyatt Sao Paulo

Freescale.com/FTF

www.Freescale.com