[1]

FreeRTOS

“It is the fate of operating systems to become free.”

— Neal Stephenson

Prof. Erich Styger

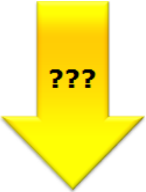
erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

March 30, 2017

1 –Infotronic

Introduction

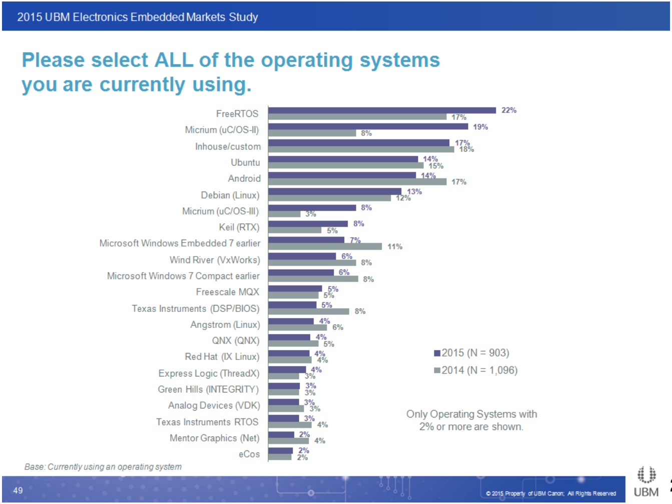
- Architecture and Licensing
- High level services
- Cortex-M interrupts
- Kernel, Tasks, Hooks Heap
- Queues, Semaphore, Mutex
- Task notification



Introduction

- `http://www.freertos.org`
- Maintained by Real Time Engineers Ltd., London (Richard Barry)
- Open Source, free-of-charge, royalty free
- $>$35 architectures, $>$113'000 downloads in 2015
- Portable, simple to learn and use
- Ecosystem and commercial supported ports available
  - **OpenRTOS**: commercial supported version
  - **SafeRTOS**: special version dedicated to safety critical systems
  - Sold Reference Manual/Tutorial Book (protected PDF with watermark, no print/copy)

UBM Embedded Market Study



[2]

[2] Source: http://tech.ubm.com

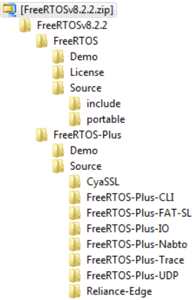
Services and Features

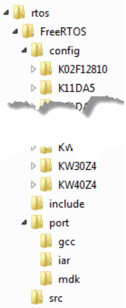
- Scheduler
- Tasks with multiple priority lists
- Dynamic memory (heap)
- Coroutines
- Message queue
- Software timer
- Semaphore and Mutex
- Event set
- Direct task notification
- Thread Local Storage pointers
- Low Power and Tickless Idle Mode
- Trace

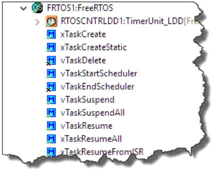
FreeRTOS Licensing

- Terms on www.freertos.org/a00114.html
- Open Source, Free of charge, no royalties
- *"The modification to the GPL is included to allow you to **distribute a combined work** that includes FreeRTOS **without being obliged to provide the source code** for proprietary components"*
- Community support: sourceforge.net/projects/freertos
- Can be used in commercial applications, need to contribute back changes in the **kernel**
- If source code is published, FreeRTOS needs to be published too
- Not allowed to publish benchmarks results without permission
- **Different** licensing terms for OpenRTOS, SafeRTOS and FreeRTOS+ parts

Distributions

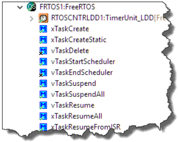


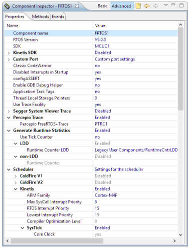




1. Default distribution: http://www.freertos.org
2. Integrated into SDK's (NXP Kinetis SDK, http://www.nxp.com/ksdk) with SDK Processor Expert component
3. Advanced Open Source FreeRTOS port and component: sourceforge.net/projects/mcuoneclipse

McuOnEclipse





- `https://sourceforge.net/projects/mcuoneclipse/`
- Latest Open Source FreeRTOS port for S08, S12(X), S12X, DSC, ColdFire V1/V2 and **Kinetis**
- Graphical configuration of `FreeRTOSConfig.h`
- Extra features: **Command line shell**, **Low Power** Timer and **Tickless Idle** mode, Percepio **Tracealizer**, Segger RTT with **SystemViewer**

Summary

- FreeRTOS is open source
- Permissible license, allows to link it with my code without affecting license
- Commercial version: **OpenRTOS** and **SafeRTOS**
- Processor Expert component
- Multiple Open Source ports available

Architecture

"It is not the beauty of a building you should look at; its the construction of the foundation that will stand the test of time."

— David Allan Coe

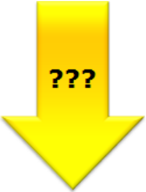
Prof. Erich Styger

erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

March 30, 2017

Architecture: Learning Goals

- Know the design philosophy
- Understand source organization
- Apply interrupt model to application



Philosophy- Kernel

- Small Kernel, **implemented in C**, compiled and linked with application
- ARM Cortex M0+ and M4(F) ports
- Kernel configuration with `#define` in **`FreeRTOSConfig.h`**
- Preemptive or cooperative scheduler mode (at compile time)
- Kernel only needs **tick interrupt** and **software interrupt**
- RTOS creates and runs in IDLE task
- Scheduler variables and task stack in dynamic memory (**heap**)
- Different selection of heap allocation (**schemes**)
- Multiple tasks with same priority

Philosophy- Ticks

operating systems need a time tick

- Tick or 'Ticks' provide time base for RTOS
- Counter in tick interrupt
- Typically 10 ms or 1 ms tick period, default max 1 kHz
- ARM: SysTick timer, or low power timer (LPTMR on Kinetis), or any periodic timer
- All RTOS time calculations are in ticks!
- Tick frequency/period considerations
  - Interrupt and system load
  - Timing precision

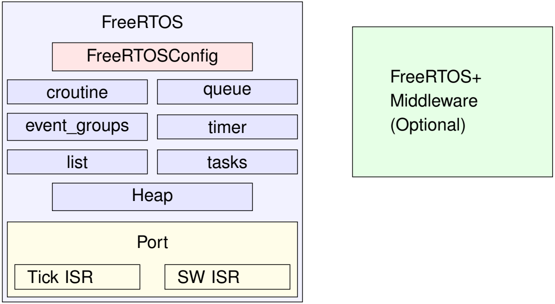
PSP

MSP

Philosophy- Tasks

- Possible **dynamic task** creation and deletion
- Task stack and scheduler data structure in dynamic memory (**heap**)
- Tasks are running with stack in the 'heap'
- Interrupts are running on 'main' stack (MSP)
- Tasks are using PSP stack pointer
- Software interrupt used to switch task context
- Tasks are (usually) staying in an endless loop
- RTOS always creates and runs IDLE task

Block Diagram



- Only 10 source plus header files for co-routine, queue, event, timer, list and tasks
- FreeRTOS Configuration Header File
- 5 different heap implementations
- Port depending on architecture and tool chain and compiler
- FreeRTOS+: different license

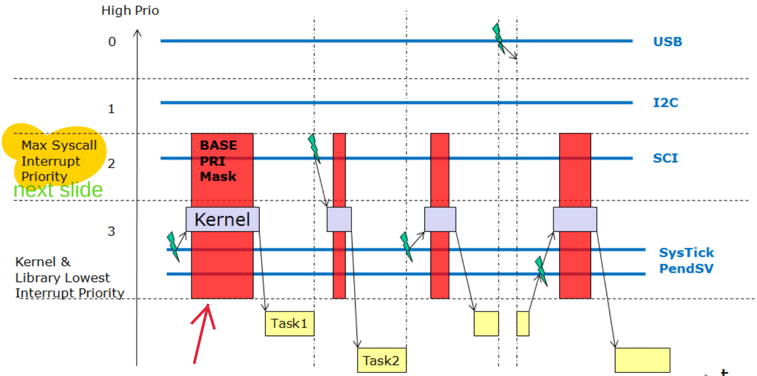
Kernel and Interrupts

- RTOS needs two interrupt sources
  - **Tick Interrupt**: `SysTick`
  - **Software Interrupt**: `SVCall`, `PendableSrvReq`
- Other interrupts: under application control
- Need to care about reentrancy
- RTOS does not protect its own data structures
  - Keep interrupts enabled in RTOS
  - Efficiency and interrupt latency
  - RTOS API functions with `FromISR` suffix (e.g. `xSemaphoreGiveFromISR`) use critical sections
  - Special consideration: interrupt nesting

with this assembly-code

trigger are executed by trigger

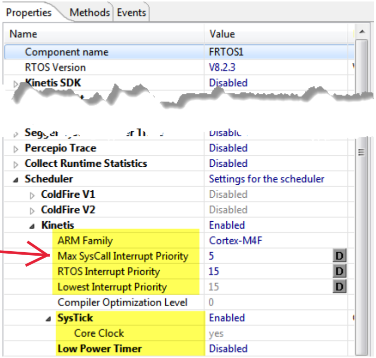
ARM Cortex-M Interrupts



M0: would disable all interrupts

- Cortex-M0+: Kernel disables global interrupts
- Cortex-M4: using `PRIMASK` to mask interrupts 0: -> it is disabled (no masking)
  - `configMAX_SYSCALL_INTERRUPT_PRIORITY`
    like BASEPRI -> check it

ARM Cortex-M Interrupts- Component Settings



view slide #17

- ARM Family and Core Settings
- `configMAX_SYSCALL_INTERRUPT_PRIORITY`
- Interrupt setting for RTOS (lowest!): SysTick, PendSV and SVCall
- Using Systick as tick timer

Architecture: Summary

- **Portable**, C, 'Port' with ASM
- Kernel only needs **tick** and **SWI**
- Kernel usually does **not block all interrupts** (blocks all on Cortex M0+!)
- ARM Cortex-M4: `configMAX_SYSCALL_INTERRUPT_PRIORITY`
- **Heap** needed for Kernel data and task stacks
- Do **not** call RTOS API from interrupts except `FromISR` API

Universal Asynchronous Receiver Transmitter UART

Kernel Control

“Must is a hard nut to crack, but it has a sweet kernel.”

— Charles Spurgeon

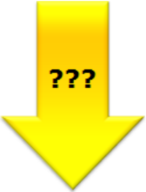
Prof. Erich Styger

erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

March 30, 2017

Kernel Control: Learning Goals

- Learn different ways to control the kernel
- Use kernel to create critical sections
- Start/stop/suspend the scheduler
- Apply it for your application



Kernel Control: Overview

- Starting and stopping the scheduler:
  `vTaskStartScheduler(),` starting
  `vTaskEndScheduler()` ending
- Kernel suspending and resuming:
  `vTaskSuspendAll(),vTaskResumeAll()`
- Passing control to one of the ready tasks:
  `taskYIELD()`

vTaskStartScheduler

```
void vTaskStartScheduler(void);
```

- Starts the scheduler (Init $\rightarrow$ Running)
- Creates IDLE task (`configMINIMAL_STACK_SIZE`, `tskIDLE_PRIORITY`)
  in 4-Byte units
  0,1,2,3.. MAX (0 is the lowest!, because task)
- Creates the optional timer daemon task.
- Runs the highest priority task
- Does not return until `xTaskEndScheduler()`

vTaskEndScheduler

```c
void vTaskEndScheduler(void);
```

- Kernel resources will be released
- Task resources (queues, semaphores) are *not* freed
- Many ports do *not* implement this function

vTaskSuspendAll

```
void vTaskSuspendAll(void);
```

- Puts the kernel from *Active* to *Suspended* state
- Prevents context switch
- Interrupts remain enabled tick interrupts are still running (keeping running)
- Can be called in a nested way

xTaskResumeAll

```
portBASE_TYPE xTaskResumeAll(void);
```

- Puts the kernel from *Suspended* to *Active* state
- return value
  - *pdTRUE*: Context switch happened
  - *pdFALSE*: no context switch or still Suspended (nested)

taskENTER_CRITCAL, taskEXIT_CRITCAL

```c
void taskENTER_CRITICAL(void);
void taskEXIT_CRITICAL(void);

void vPortEnterCritical(void) {
  portDISABLE_INTERRUPTS();
  uxCriticalNesting++;
}

void vPortExitCritical(void) {
  uxCriticalNesting--;
  if (uxCriticalNesting == 0) {
    portENABLE_INTERRUPTS();
  }
}
```

interrupts are disabled even the timer interrupts

do your stuff in there, but do it fast

- Counter used to count nesting
- Some ports always enable interrupts when count==0
- Do *not* call FreeRTOS API functions in CS!

taskDISABLE_INTERRUPTS, taskENABLE_INTERRUPTS

```
void taskDISABLE_INTERRUPTS(void);
void taskENABLE_INTERRUPTS(void);
#define taskDISABLE_INTERRUPTS()    \
  portDISABLE_INTERRUPTS()
#define portDISABLE_INTERRUPTS()    \
  portSET_INTERRUPT_MASK()
#define portSET_INTERRUPT_MASK()    \
  __asm volatile("cpsid i")
#define portCLEAR_INTERRUPT_MASK()  \
  __asm volatile("cpsie i")
```

for M0!

- *Globally* enables and disables interrupts (macros)
  - Cortex-M0+: all interrupts
  - Cortex-M4: up to `PRIMASK`, `configMAX_SYSCALL_INTERRUPT_PRIORITY`

taskYIELD

```
#define taskYIELD()   portYIELD()
#define portYIELD()   vPortYieldFromISR()
 yield causes an interrupt
void vPortYieldFromISR(void) {
  /* Set a PendSV to request a context switch. */
  *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET_BIT;
  /* Barriers are normally not required but do ensure
       the code is completely within the specified
     behavior for the architecture. */
  __asm volatile("dsb");   barrier instructions: pipline flashing mememory
  __asm volatile("isb");
}
```

- Request a context switch with *Software Interrupt*
- Required for cooperative multitasking
- Typically implemented as macro

Kernel Control: Summary

- FreeRTOS API to control the kernel
- Different states of kernel (init, running, suspended)
- How to start/stop the scheduler
- Suspending/resuming tasks with interrupts enabled
- Creating Scheduler Critical Sections
- Disabling/Enabling interrupts
- Yielding

Tasks

"It seems essential, in relationships and all tasks, that we concentrate only on what is most significant and important."

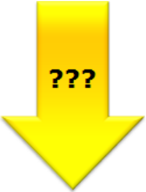
— Soren Kierkegaard

Prof. Erich Styger

erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

Tasks: Learning Goals

- Understand preemptive and cooperative mode
- Know FreeRTOS priority numbering
- Overview of task states and transitions
- Apply different options for time slicing
- Understand how the IDLE task works

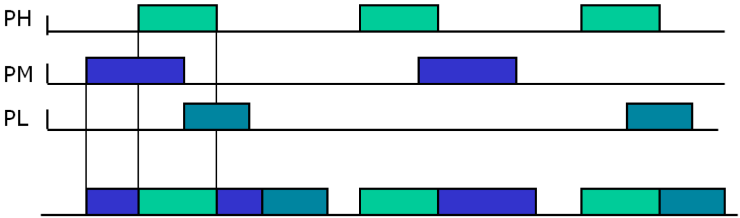


Tasks: Overview

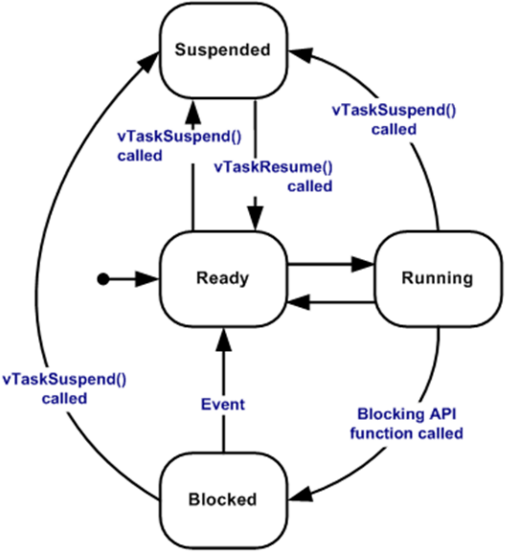
- FreeRTOS task system: Task transitions
- Kernel and Task Control API
  - Scheduler start and stop
  - Task creating, deleting, suspending and resuming
- Scheduling and Idle Task

Priority-based Preemptive Scheduling

- `configUSE_PREEMPTION` configures preemptive scheduling
- Always run the highest-priority ready task
- Priorities from 0 (lowest) to (`configMAX_PRIORITIES - 1`)
- Multiple tasks with the same priority: time-slicing
- High priority tasks should not starve lower priority tasks



Task States



you are waiting on something -> blocked!

Time Slicing

- `configUSE_TIME_SLICING` is default mode: time slicing between highest ready tasks with same priority at tick interrupt time.
- Processing time given to ready task with highest priority
- Tasks with the same priority will *time slice*

IDLE Task

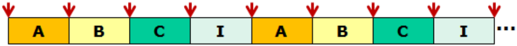
- IDLE task always created at scheduler start
- Priority: `tskIDLE_PRIORITY`
- Stack size: `configMINIMAL_STACK_SIZE`
- Runs if no other task needs to run
- Frees memory of deleted tasks
  - No IDLE starving if using `vTaskDelete()`
  - Otherwise: IDLE starving is fine
- Calls Idle Task Hook
  - Application functionality in idle hook/task
  - Common use: enter power saving mode

IDLE Task

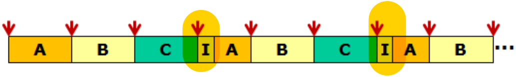
```
static void prvIdleTask(void *pvParameters) {
  for(;;) {                         every task has a parameter
    RemoveDeletedTasksFromList();
    if (!configUSE_PREEMPTION) {
      taskYIELD();
    } else if (configIDLE_SHOULD_YIELD) {
      if (NofReadyTasks(tskIDLE_PRIORITY)>1) {
        taskYIELD();
      }           how much stack to add?
    }
    IdleHook();
  } /* for */
}
```

- Yields if using cooperative scheduler mode
- `configIDLE_SHOULD_YIELD`: yield if ready tasks are available

IDLE Task: configIDLE_SHOULD_YIELD



- `configIDLE_SHOULD_YIELD` for preemptive scheduler
- Time slicing will distribute time equally among same priority tasks
- CPU cycles wasted in IDLE task



- With enabled `configIDLE_SHOULD_YIELD` IDLE task gives CPU time to tasks
- No exact time slicing any more

Blinky Task- Task Creation

```
BaseType_t res;                                  v: means returns nothing
xTaskHandle taskHndl;
res = xTaskCreate(BlinkyTask, /* function */ x: means returns something
        "Blinky", /* Kernel awareness name */
        configMINIMAL_STACK_SIZE+50, /* stack */
        (void*)NULL, /* task parameter */
        tskIDLE_PRIORITY, /* priority */
        &taskHndl /* handle */
      );
if (res!=pdPASS) { /* error handling here */ }
```

- Task function and debug name
- Stack size for task and priority
- Optional task parameter (`pvParameters`) or `NULL`
- Optional task handle or `NULL`
- Check error code (out of memory?)

Blinky Task

```
static void BlinkyTask(void *pvParameters) {
  for(;;) {
    LED_Neg();
  }
}
```

- Tasks are normal functions
- start parameter `pvParameters` pointer assigned with `xTaskCreate()`
- Endless loop, does not leave function (except task deletes itself)
- BlinkyTask runs until preempted

Blinky Task- vTaskDelay

```
static void BlinkyTask(void *pvParameters) {
  for(;;) {
    LED_Neg();
    vTaskDelay(50/portTICK_PERIOD_MS);
  }                    1 for 1kHz
}                      2 for 500Hz
```

- Avoid starving other task
- `portTICK_PERIOD_MS`: time between two ticks
- `vTaskDelay()` suspends task for given ticks
- Suspends number of ticks from current tick count

Blinky Task- vTaskDelayUntil

```
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime,
    const TickType_t xTimeIncrement);
```

```
static void BlinkyTask(void *pvParameters) {
  TickType_t xLastWakeTime = xTaskGetTickCount();
  for(;;) {
    LED_Neg();
    vTaskDelayUntil(&xLastWakeTime, 50/portTICK_PERIOD_MS);
  }
}
```

- `xTaskGetTickCount()` returns current tick counter
- `vTaskDelayUntil()` can delay from previous tick counter, independent of task overhead
- `xLastWakeTime` updated by kernel

Task Control: Overview

- Delaying a task:
  `vTaskDelay(), vTaskDelayUntil()`
- Changing the priority at runtime:
  `uxTaskPriorityGet(), vTaskPrioritySet()`
- Task suspending and resuming:
  `vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()`

uxTaskPriorityGet

```c
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle
        pxTask);
```

- Returns the priority of a task
- NULL argument: priority of calling task
- Call only from a task context

```c
xTaskCreate(vTaskCode, "MyTask",
    configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY,
    &xHandle);
if (uxTaskPriorityGet(xHandle)!=tskIDLE_PRIORITY) {
  /* The task priority has been changed? */
}
```

or 0 if you want to check your own prio

vTaskPrioritySet

```
void vTaskPrioritySet(xTaskHandle pxTask, unsigned
    portBASE_TYPE uxNewPriority);
```

- Changes the priority of a task
- NULL argument: priority of calling task
- Call only from a task context

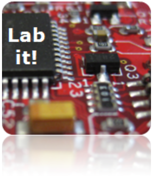
```
xTaskCreate(vTaskCode, "MyTask",
    configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY,
     &xHandle);
vTaskPrioritySet(xHandle, tskIDLE_PRIORITY+1);
```

Tasks: Summary

- Preemptive and non-premptive scheduling
- Task time slicing
- `configMINIMAL_STACK_SIZE` used for IDLE task
- `tskIDLE_PRIORITY` is zero and the lowest priority
- Tasks usually run in an endless loop
- Task creation can call out-of-heap error hook
- `vTaskDelay()` and `vTaskDelayUntil()`

Tasks: Lab Task

- Create 'blinky' FreeRTOS tasks
- Task creation outside of scheduler
- Task creation from task
- Using task handle
- Using task startup parameter
- Using `vTaskDelay()` and `vTaskDelayUntil()`



Hooks

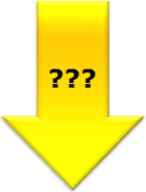
“Do not bite at the bait of pleasure, till you know there is no hook beneath it.”

— Thomas Jefferson

Prof. Erich Styger

erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

Hooks- Goals



- Understand FreeRTOS hook concept
- Know the different hooks
- Apply and use hooks in application

Hooks: Overview

- RTOS calls optional application defined callbacks (or hooks)
- Hook function implemented in application
- Available hooks
  - **Idle Hook**: called whenever the system is idle
  - **Tick Hook**: called for every system time tick
  - **Malloc Failed Hook**: called if allocation failed
  - **Stack Overflow Hook**: called if stack overflow has been detected

Idle Hook

```
void vApplicationIdleHook(void) {
  /* Called whenever the RTOS is idle (from the IDLE task ) */
  CPU_EnterLowPowerMode(); /* wait for interrupt */
  /* here an interrupt woke us up */
}
```

- Enabled with `configUSE_IDLE_HOOK`
- Called from the IDLE task, whenever the system no user task is running
- Ideal place to go into low power mode
- Toggle LED/Pin: activity shows idle activity ⇒ Low Power
- Idle hook function *not* allowed to call blocking API functions!

Blocking
vTaskDelay() -> please me als IDLE task don't do sth for ...ms
SemaphoreTake()

Tick Hook

```
void vApplicationTickHook(void) {
  /* Called for every tick interrupt */
  PIN_Toggle(); /* debug tick interrupt */
}
```

- Enabled with `configUSE_TICK_HOOK`
- Called from tick interrupt
- Can be used instead of a dedicated timer running at tick frequency
- Executed in interrupt context, keep it short!
- Only `FromISR()` RTOS API functions can be called

Malloc Failed Hook

```c
void vApplicationMallocFailedHook(void) {
  /* Called if a call to pvPortMalloc() fails because
     there is insufficient free memory available in the
     FreeRTOS heap.  pvPortMalloc() is called internally
     by FreeRTOS API functions that create tasks,
     queues, software timers, and semaphores. The size
     of the FreeRTOS heap is set by the
     configTOTAL_HEAP_SIZE configuration constant in
     FreeRTOSConfig.h. */
  taskDISABLE_INTERRUPTS();
  for(;;) {} /* stop for debugging */
}
```

- Enabled with `configUSE_MALLOC_FAILED_HOOK`
- Called if memory allocation failed

Stack Overflow Failed Hook

very useful one!

```c
void vApplicationStackOverflowHook(xTaskHandle pxTask, char *pcTaskName) {
  /* This will get called if a stack overflow is detected during the context switch. Set configCHECK_FOR_STACK_OVERFLOWS to 2 to also check for stack problems within nested interrupts, but only do this for debug purposes as it will increase the context switch time. */
  taskDISABLE_INTERRUPTS();
  for(;;) {} /* stop for debugging */
}
```

- Enabled with `configCHECK_FOR_STACK_OVERFLOWS`
- Overflow check adds run-time cost
- Method 1: Check SP at task swap time not really faile-save
- Method 2: Method 1 + checking last 16 bytes pattern

check if the SP in the stack?

Hooks: Summary

- Concept: hooks are used as events to notify the application
- Hooks are optional
- Enabled/disabled in `FreeRTOSConfig.h`
  - **Idle Hook**: called whenever the system is idle
  - **Tick Hook**: called for every system time tick
  - **Malloc Failed Hook**: called if malloc failed
  - **Stack Overflow Hook**: called if stack overflow has been detected
- Error hooks: disable interrupts and halt system

Heap Memory

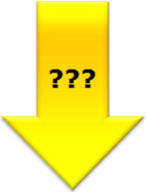
“A good memory is surely a compost heap that converts experience to wisdom, creativity, or dottiness; not that these things are of much earthly value, but at least they may keep you amused when the world is keeping you locked away or shutting you out.”

— Michael Leunig

Prof. Erich Styger

Heap Memory: Learning Goals

- Knowing Memory allocation in FreeRTOS
- Avoid memory fragmentation
- Choosing the right memory scheme
- Using dynamic memory allocation



Heap Memory: Overview

- Dynamic memory used by RTOS: `configTOTAL_HEAP_SIZE`
  - Size in bytes
  - Tasks stack space
  - RTOS internal structures: TCB Task Control Block
  - Semaphores, Mutex, Queues
  - Application memory through `pvPortMalloc()`
- Availability: not always available with tool chain
- Efficiency: tuned for small code size
- Thread safe: to be used from multiple threads
- Deterministic: timing guaranteed within error margin
- Multiple allocation 'Schemes', one selected at configuration time

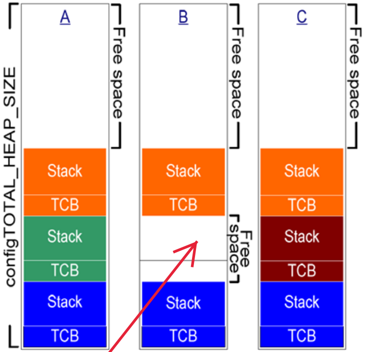
Heap Schemes

- `configFRTOS_MEMORY_SCHEME`
- Custom memory allocation can be added
- 5 predefined allocation 'Schemes'

runtime

1. Allocation only, deterministic can't free it!
2. No block merge, not deterministic, heap fragmentation
3. Wrapper to standard `malloc()/free()`, not deterministic
4. Coalesces free blocks, not deterministic rather using this one
5. Multiple memory segments, coalesces blocks

Memory Fragmentation: Scheme 2



- Blocks and tasks can be deleted
- Memory blocks are not combined
- Possible memory fragmentation
- Do not use for random allocation/free sequences

if you kill the task and then renew the task with the same size -> ok! otherwise it doesn^t workd

Heap Example

```c
void *pvPortMalloc(size_t xWantedSize);
void vPortFree(void *pv);
size_t xPortGetFreeHeapSize(void);
```

- Check return value if not NULL

```c
void foo(void) {
  uint8_t *bufP; /* pointer to buffer */

  bufP = (uint8_t*)pvPortMalloc(sizeof("Hello"));
  if (bufP==NULL) {
    for(;;); /* ups! Out of memory? */
  }
  (void)strcpy(bufP, "Hello"); /* copy data */
  /* do something with data */
  vPortFree(bufP); /* release memory */
}
```

Heap Memory: Summary

- Different heap schemes in FreeRTOS
- Alloc only, no merge, standard malloc, merge, multiple regions
- Problem of determinism and fragmentation
- Used for task stacks and RTOS data structures: queues, semaphore, mutex
- Used by application to allocate/release dynamic memory

4

Queues

“An Englishman, even if he is alone, forms an orderly queue of one.”

— George Mikes

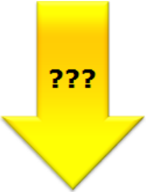
Prof. Erich Styger

erich.styger@hslu.ch

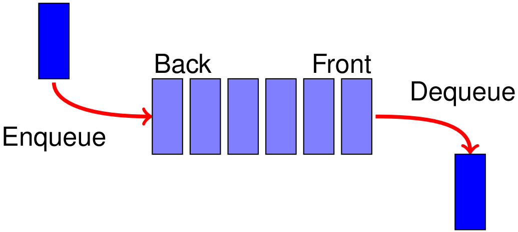
Lucerne University of Applied Sciences and Arts

Queues: Learning Goals

- Understand queues in FreeRTOS
- How to use queues
- Using FIFO and LIFO queues
- How to debug queues



Queues: Overview

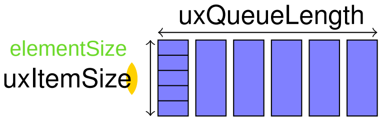


- List of items, allocated in **Heap**
- FIFO: **F**irst **I**n - **F**irst **O**ut
- LIFO/Stack: **L**ast **I**n - **F**irst **O**ut
- Queues have ability to block caller if queue full or empty
- RTOS can resume tasks which are blocked on a queue
- FreeRTOS: Enqueue and Dequeue *by copy* operation

queues are

Creating and Deleting Queues

```
xQueueHandle xQueueCreate(
  unsigned portBASE_TYPE uxQueueLength,
  unsigned portBASE_TYPE uxItemSize
);
void vQueueDelete(xQueueHandle xQueue);
```



- `xQueueCreate()`: fixed queue size at creation time
- Returns Queue handle, check for NULL!
- `uxQueueLength`: maximum number of items in queue
- `uxItemSize`: element item size in bytes

Inspecting and Removing Items

```
BaseType_t xQueueReset(xQueueHandle xQueue);
BaseType_t xQueuePeek(
  xQueueHandle xQueue,
  void *pvBuffer,
  portTickType xTicksToWait);
BaseType_t xQueueReceive(
  xQueueHandle xQueue,
  void *pvBuffer,
  portTickType xTicksToWait);
```

everything is in ticks, can specify the duration of the timeout

directly get returned

- `xQueueReset()`: Clear queue
- `xQueuePeek()`: Inspecting item without removing
- `xQueueReceive()`: Remove item in front the queue
- `pvBuffer`: pointer where to copy the item
- `xTicksToWait`: Waiting ticks/time
  - 0: polling
  - `portMAX_DELAY`: wait forever

Adding Items

```
BaseType_t xQueueSendToBack(
  xQueueHandle xQueue,
  const void *pvItemToQueue,
  portTickType xTicksToWait);
BaseType_t xQueueSendToFront(
  xQueueHandle xQueue,
  const void *pvItemToQueue,
  portTickType xTicksToWait);
```

- `xQueueSendToBack()`: FIFO operation
- `xQueueSendToFront()`: Stack/LIFO operation
- `xTicksToWait`: Waiting time in case queue is full
- returns `pdPASS` or `errQUEUE_FULL`

Queue Registry

```
void vQueueAddToRegistry(
  QueueHandle_t xQueue,
  const char *pcQueueName);
void vQueueUnregisterQueue(QueueHandle_t xQueue);
```

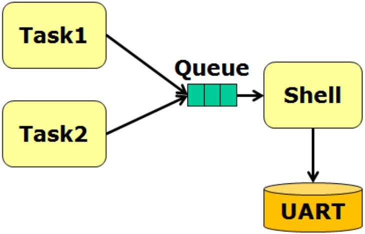
- `configQUEUE_REGISTRY_SIZE`: number of queues for registry maximum number of queues
- `vQueueAddToRegistry()`: make queue name known
- `vQueueUnregisterQueue()`: remove queue name from list

Queue List (FreeRTOS)

#	Queue Name	Address	Len...	Item Size	# T...	# R...	Queue Type
1	RefStartStopSem	0x20000848	0/1	Empty	0	0	Binary Semaphore
2	RefSem	0x200008a0	1/1	Empty	0	0	Mutex
3	RadioRxMsg	0x20001138	0/6	0x22 (34 B)	0	0	Queue
4	RadioTxMsg	0x20001260	0/6	0x22 (34 B)	0	0	Queue
5	RxStdInQ	0x20001388	0/48	0x1 (1 B)	0	0	Queue

maximum of items

Example: Shell Queue



we need to protect the UART, because two Tasks have access on it

- Peripheral sharing
- Inter-Process Communication (IPC)
- Consumer-Producer Pattern
- Items: characters or messages

Queues: Summary

- Message passing, data buffering and IPC
- Queues are allocated in heap memory
- Fixed at creation time
- Two-Dimensional arrays of data
- Items 'by value', *not* 'by reference' items are stored by value
- Queuing in blocking and polling ways

Queues: Lab Task, Queue Quiz 1

```c
#define QUEUE_LENGTH      5
#define QUEUE_ITEM_SIZE   sizeof(uint8_t*)

void QUEUE_SendMessage(const uint8_t *msg) {
  uint8_t *ptr;
  size_t strSize = strlen(msg)+1;

  ptr = FRTOS1_pvPortMalloc(strSize);
  (void)UTIL1_strcpy(ptr, strSize, msg);
  if (FRTOS1_xQueueSendToBack(
        queueHandle, ptr, portMAX_DELAY)!=pdPASS)
  {
    for(;;){} /* ups? */
  }
}
```

we need to have the address of the ptr, put there an address (&)

Queues: Lab Task, Queue Quiz 2

```
static xQueueHandle queue;

void QUEUE_SendMessage(const uint8_t *msg) {
  FRTOS1_xQueueSendToBack(queue, msg, 0);
}

const uint8_t *QUEUE_ReceiveMessage(void) {
  uint8_t *buffer;

  FRTOS1_xQueueReceive(queue, buffer, 0);
  return buffer;
}

void QUEUE_Init(void) {
  queue = FRTOS1_xQueueCreate(3, 3);
}
```

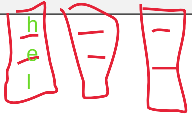
using polling

it copy just three characters

-> queue three by three

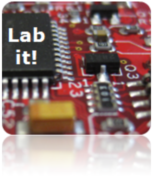
if msg = "hello0"

queue by: three by three



Queues: Lab Task

- Lab Assignment: **Queues**
- Implement message passing between tasks with queues
- Multiple tasks use queue to send data to shell task
- Evaluate and discuss 'single char' and 'message' queue variants
- Apply combination of polling and queuing with timeouts



Semaphore and Mutex

"Quantum Mechanics and General Relativity are both accepted as scientific fact even though they're mutually exclusive."

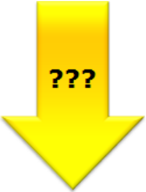
— Roy H. Williams

Prof. Erich Styger

erich.styger@hslu.ch
Lucerne University of Applied Sciences and Arts

Semaphore and Mutex: Learning Goals

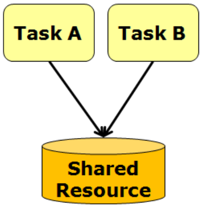
- Understand why using *Synchronization Primitives*
- Identify problems of *Priority Inversion* and *Deadlocks*
- Apply *Priority Inheritance* and *Priority Ceiling* protocols
- Use FreeRTOS Semaphore and Mutex in applications

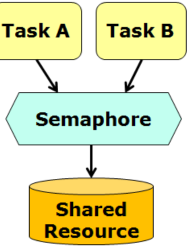


Semaphore and Mutex- Overview

- RTOS provides ways for communication and synchronization
- **Semaphore**: Binary and Counting Semaphore, Mutex
- **Problems**: Priority Inversion, Deadlocks
- **Protocols**: Priority Inheritance, Priority Ceiling
- **Application**: Resource guard/protection/synchronization, IPC

Access to Shared Resource





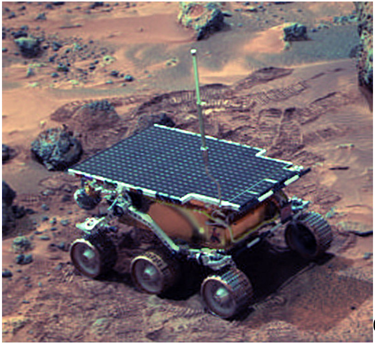
- Multiple tasks or threads need access to shared resource
- Problems: coordinated access, serialization, critical sections
- Need for *Synchronization*

Synchronization

- *Critical Section*
  - Program section, to be executed mutually exclusively
- *Semaphore*
  - One of synchronization primitives
  - Reentrancy, Critical Section, Atomic processing
    can help to implement reentrancy
- *Mutual Exclusion* or *Mutex*
  - Special type of (binary) semaphore
  - Only one task/thread/code will be in critical section
- Notation
  - *Lock()*: $?_x \Rightarrow L_x$
  - *Unlock()*: U_x

Sojourner Mars Rover

- Mars Pathfinder mission with Sojourner rover
- Launched 4. Dec. 1996, landed 4. Jul. 1997
- Problem: Occasional resets

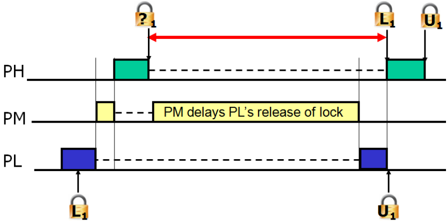

[6]

[6] Source: Wikipedia

Priority Inversion

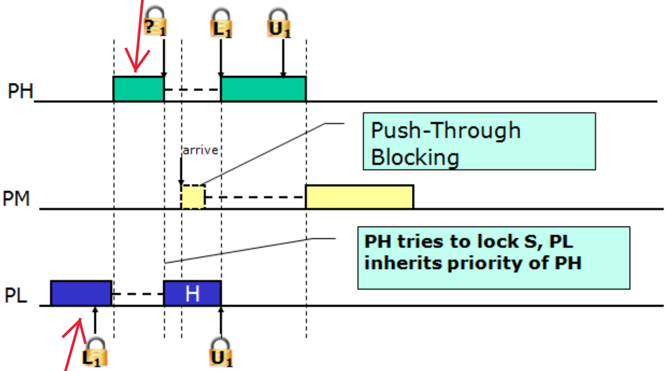
priority inversion is a problem

- *Priority Inversion*
  - Higher priority task blocked by lower priority task
- *Indefinite Priority Inversion*
  - Possible with medium priority task delaying lower priority task (which blocks high priority task)
- *Solution: Priority Inheritance*
  - Lower-priority task inherits priority of any higher-priority task pending on resource
  - At resource release: task priority change ends



Priority Inheritance Protocol

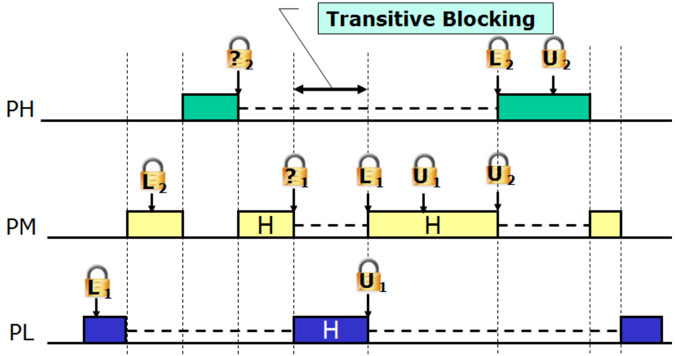
this one wants it



this one has the resouce (NASA using this semphare taking and remove)

- Task holding resource (PL) inherits priority of requesting task (PH)
- *Push-Through Blocking*: task is blocked by resource usage not of its own

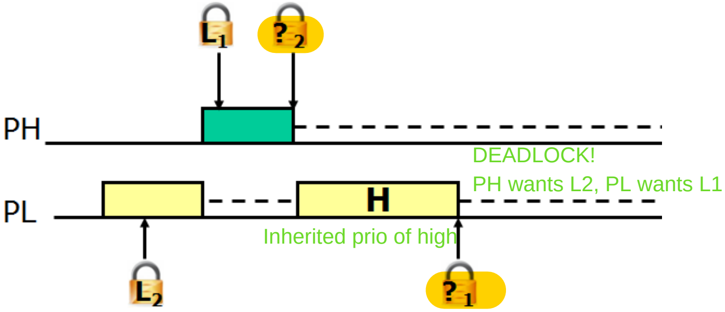
Nested Priority Inheritance



- Nested resource usage: $L_2 \rightarrow L_1 \rightarrow U_1 \rightarrow U_2$
- *Transitive Blocking*: caused by not-directly involved semaphore, accessed in a *nested* way by blocking tasks.

Deadlocks

- Priority Inheritance does *not* avoid Deadlocks
- Typical deadlock: nested resource usage



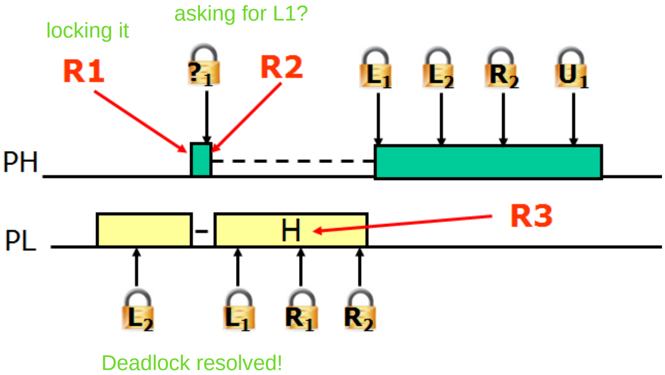
it's not connected to the ressource

Priority Ceiling (Deckel)

- Problem with deadlocks
  - Nested resources
  - Task gets resource from a class of resources which can be hold by another task of lower/same prio
- Solution with Priority Ceiling
- Resource has *Ceiling Priority*: Priority of highest task priority using it
- *Priority Ceiling* Rules
  - R1: normal scheduling (higher priority task preempts lower priority task)
  - R2: only get resource if task priority is **higher** than the ceiling of all resources **currently hold by other tasks**
  - R3: normal Priority Inheritance

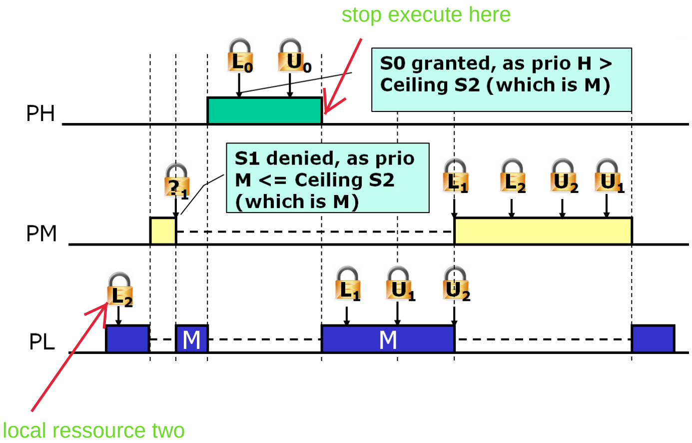
Priority Ceiling

- Priority Ceiling for both resources: H
- R1: Normal preemption
- R2: Ceiling of 1 is H, PH is not higher than ceiling of all resources currently hold (1, which is H)
- R3: Normal Priority Inheritance



Priority Ceiling

- Nested/crossing resource usage
- Priority Ceiling: $S_0 \Rightarrow H$; $S_1 \Rightarrow M$; $S_2 \Rightarrow M$



FreeRTOS Semaphore and Mutex

- FreeRTOS *Semaphore*
  - Does **NOT** implement Priority Inheritance mechanism
  - Binary (single token) and counting (multiple token) semaphore
  - Used for synchronization/IPC, does *not* have to be released
  - `xSemaphoreCreateBinary()`, `xSemaphoreCreateCounting()`
  - `xSemaphoreTake()`, `xSemaphoreGive()`, `xSemaphoreDelete()`
- FreeRTOS *Mutex*
  - **Implements** Priority Inheritance mechanism
  - Normal and **recursive** (can 'take' multiple times from same task)
  - Used for resource locking, always has to be released
  - `xSemaphoreCreate`**`Mutex`**`()`, `vSemaphoreCreateRecursive`**`Mutex`**`()`
  - `xSemaphoreTake()`, `xSemaphoreGive()`, `xSemaphoreDelete()`

Creating Binary Semaphore

```
SemaphoreHandle_t xSemaphore;

xSemaphore = xSemaphoreCreateBinary();
if (xSemaphore == NULL) {
  /* Failed! Not enough heap memory?. */
} else {
  /* The semaphore can now be used. Calling
     xSemaphoreTake() on the semaphore will fail until
     the semaphore has first been given. */
}
```

the box is empty at this moment

- Binary: one or zero token
- Creating Semaphore creates handle
- Check for NULL
- Semaphore token is not 'given' at creation time

Queues with no Data

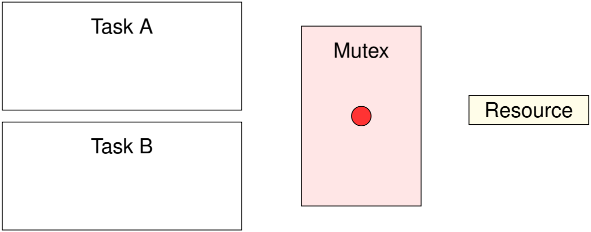
```
#define xSemaphoreCreateBinary(xSemaphore) \
  xQueueGenericCreate((UBaseType_t)1,
       semSEMAPHORE_QUEUE_ITEM_LENGTH,
       queueQUEUE_TYPE_BINARY_SEMAPHORE)

#define xSemaphoreGive(xSemaphore) \
  xQueueGenericSend((xQueueHandle)xSemaphore, NULL, \
  semGIVE_BLOCK_TIME, queueSEND_TO_BACK)

#define xSemaphoreTake(xSemaphore, xBlockTime) \
  xQueueGenericReceive((xQueueHandle)xSemaphore, NULL, \
  xBlockTime, pdFALSE)
```

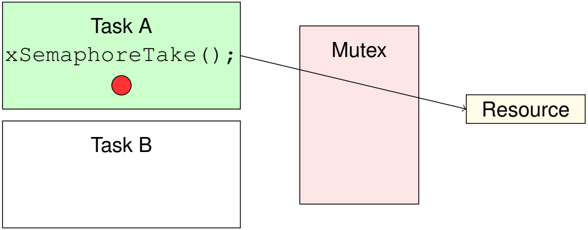
- API calls implemented as macros
- Macro `semSEMAPHORE_QUEUE_ITEM_LENGTH` is 0
- Macro `semGIVE_BLOCK_TIME` is 0

Mutex Example



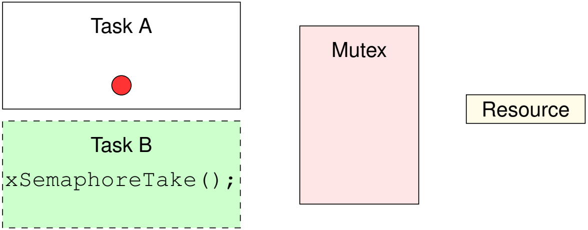
- Two Tasks: A and B
- Access to shared resource guarded with Mutex

Mutex Example



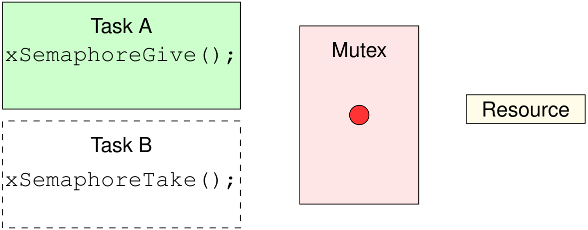
- `xSemaphoreTake()` to get resource
- Task A can access resource

Mutex Example



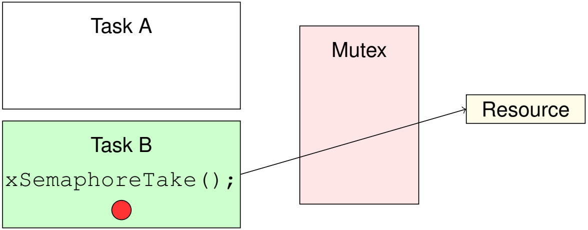
- Task B gets scheduled
- `xSemaphoreTake()` blocks Task B

Mutex Example



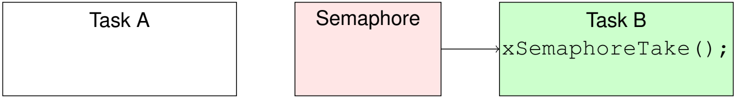
- Task A returns lock on resource with `xSemaphoreGive()`
- Scheduler can resume Task B

Mutex Example



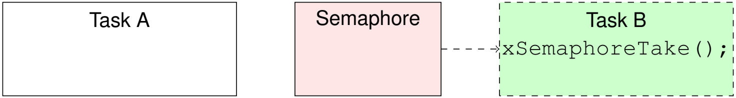
- Task B resumed by scheduler
- Call to `xSemaphoreTake()` succeeds
- Task B can access resource

Binary Semaphore Example



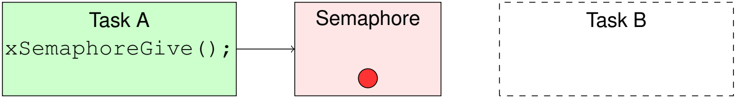
- Task B requests semaphore
- Blocking: `xSemaphoreTake()` used with 'wait forever' timeout

Binary Semaphore Example



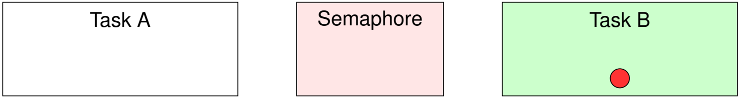
- Semaphore not available
- Kernel suspends Task B

Binary Semaphore Example



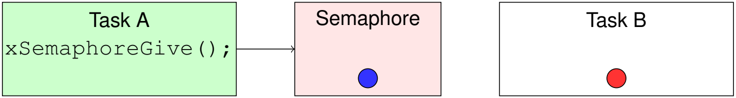
- Task A gives Semaphore
- Semaphore contains item

Binary Semaphore Example



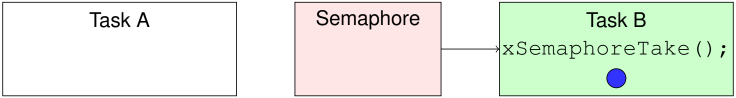
- Task B was suspended and blocked
- Advent of semaphore resumes Task B

Binary Semaphore Example



- Task B processing first message
- Scheduler could resume Task A
- Task A can produce new token while Task B suspended

Binary Semaphore Example



- Task B receives second semaphore
- Task B not blocked because semaphore is available

Semaphore and Mutex: Summary

- Semaphore and Mutex are Queues with no data
- FreeRTOS Semaphore: simple tokens
- Problems of Priority Inversion and Deadlock
- Priority Inheritance Protocol
- Priority Ceiling Protocol
- FreeRTOS Mutex: Priority Inheritance protocol
- Binary, counting and recursive

Semaphore and Mutex: Lab Task

- Lab Assignment: **Sem and Mutex**
- Implement 'master' and 'slave' tasks
- Implement simple IPC between two tasks with binary semaphore
- Discuss semaphore/mutex creation before and after scheduler started