FreeRTOS Hands-On

Objectives

- Understand RTOS concepts
- Apply the taught concepts using FreeRTOS
- Explain some design concepts based on RTOS

Prerequisites

- Good C knowledge
- Embedded SW design knowledge

Notes

- Ask any time.
- Cell phones silent, please!
- Go in/out without permission.

Reference(s)

- https://www.freertos.org/
- FreeRTOS on STM32 training MOOC

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Real-Time System = Correct Functions @Correct Time

- They can be:
  - Hard: Correct time is unnegotiable
  - Soft: Correct time can be tolerated

- Factors determining real-time behavior of a system are:
  - Determinism (predictability)
  - Responsiveness

Importance of Real-Time

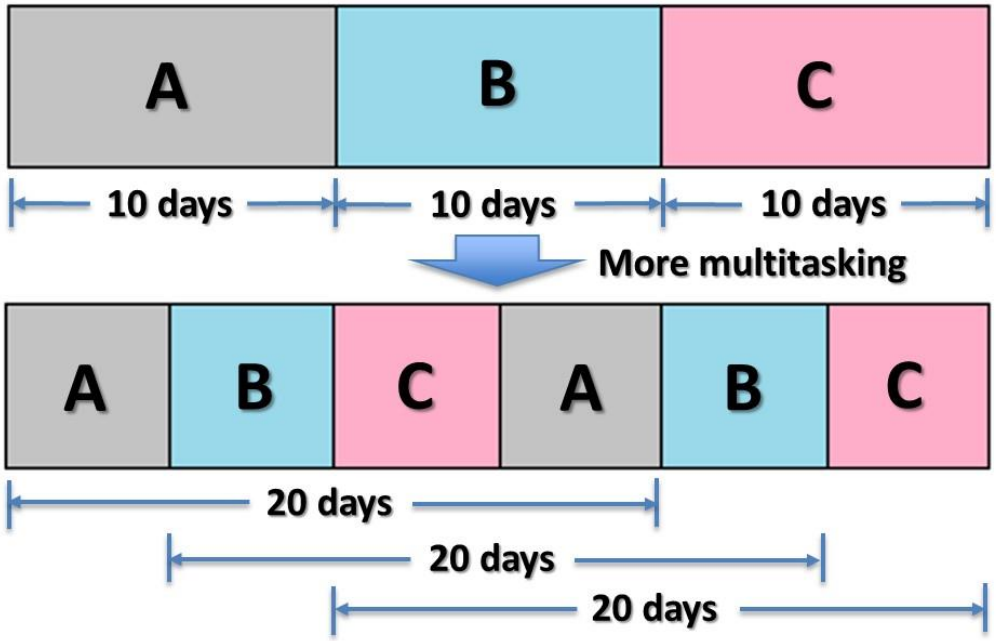
- Determinism = Predictability
- Responsiveness = Speed of response

Multitasking

- Task = Job
- Concurrency = Executing separate tasks simultaneously
- Multitasking = Executing separate tasks (seemingly) simultaneously

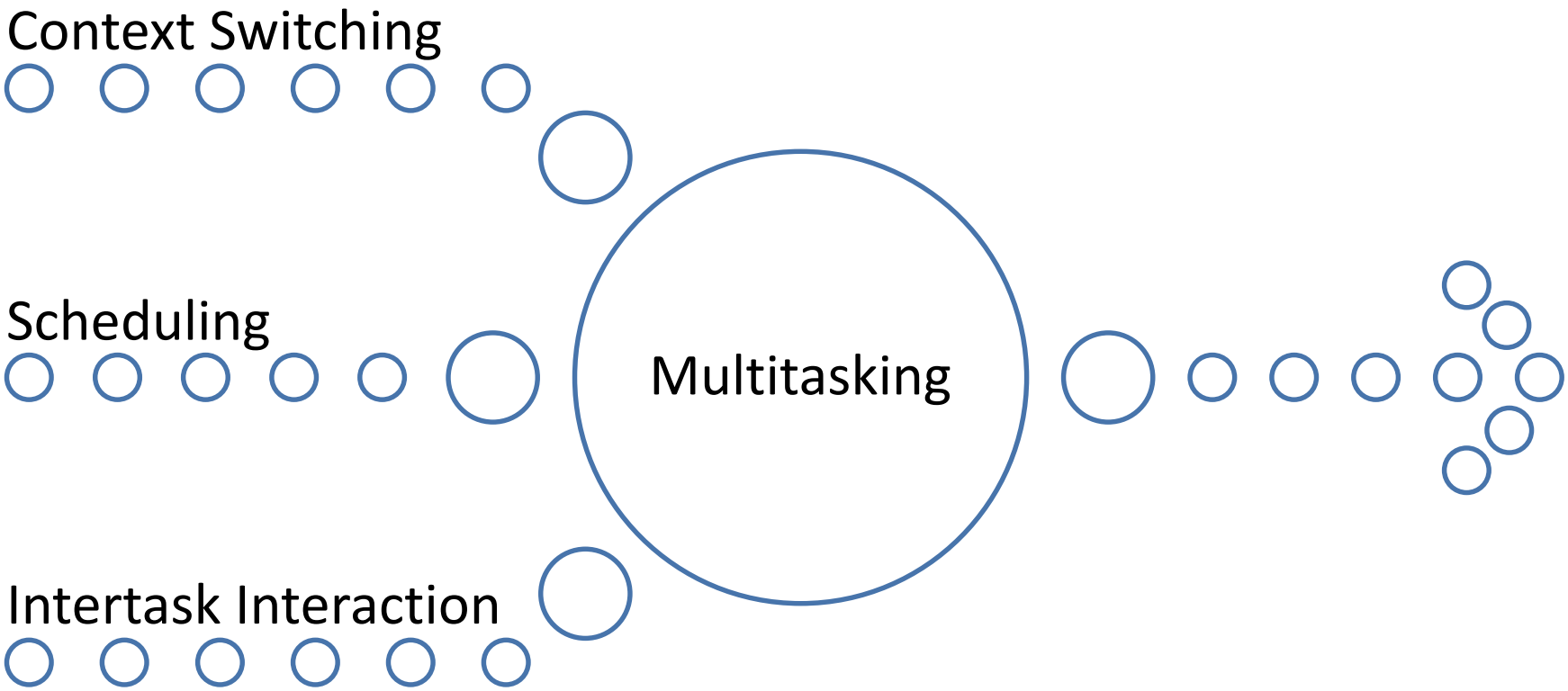
Multitasking Types

- Cooperative Non-preemptive multitasking

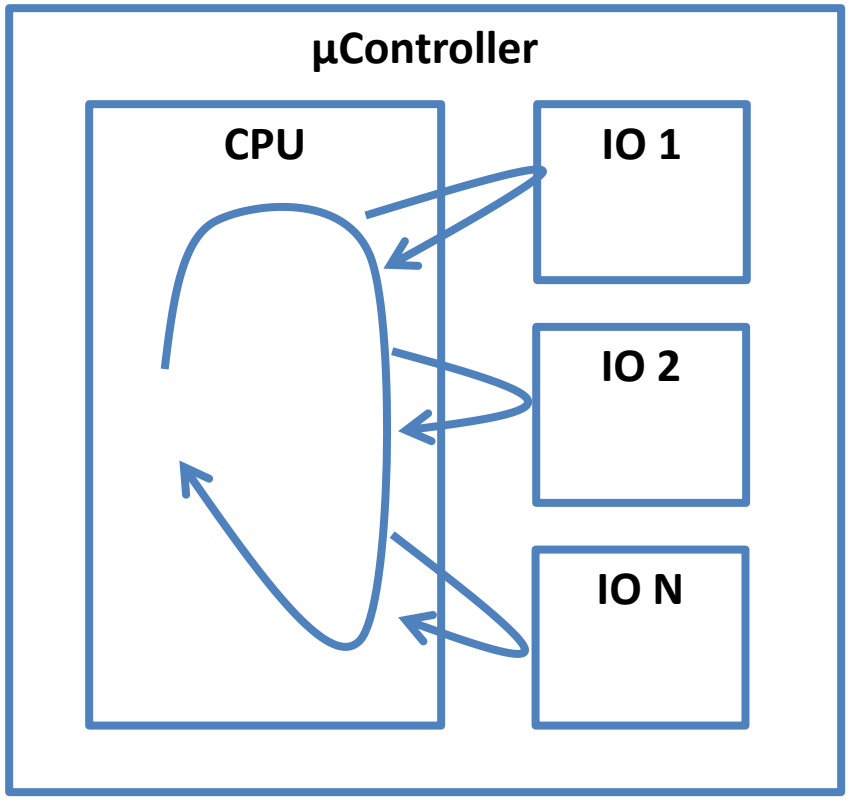


- Preemptive multitasking

Multitasking Requirements



Cyclic Executive - The Super Loop




```
int  main (void){

    /* Initialize System */

    while (1) {
        /* Periodic Tasks */
        ADC_Read();
        SPI_Read();
        USB_Packet();
        LCD_Update();
        Audio_Decode();
        File_Write();
    }
}
```


Multi-Rate Cyclic Executive - The Super Loop

- Not all tasks running @ same rate.
- Some tasks need higher rate.
  - Usage of counters
- Some tasks need lower rate.
  - Repetitive task calls

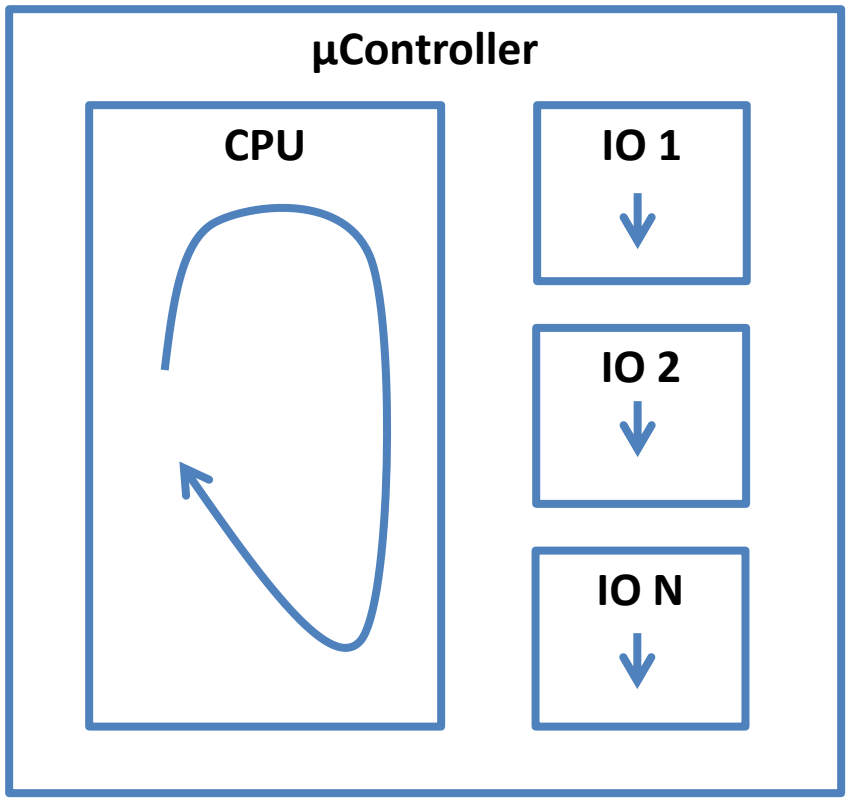
Cyclic Executive - The Super Loop Revisited

- Pros
  - Simple
  - Minimal HW resources
  - Highly portable
- Cons
  - Inaccurate timing
  - High power consumption

Super Loop as a Design Tool

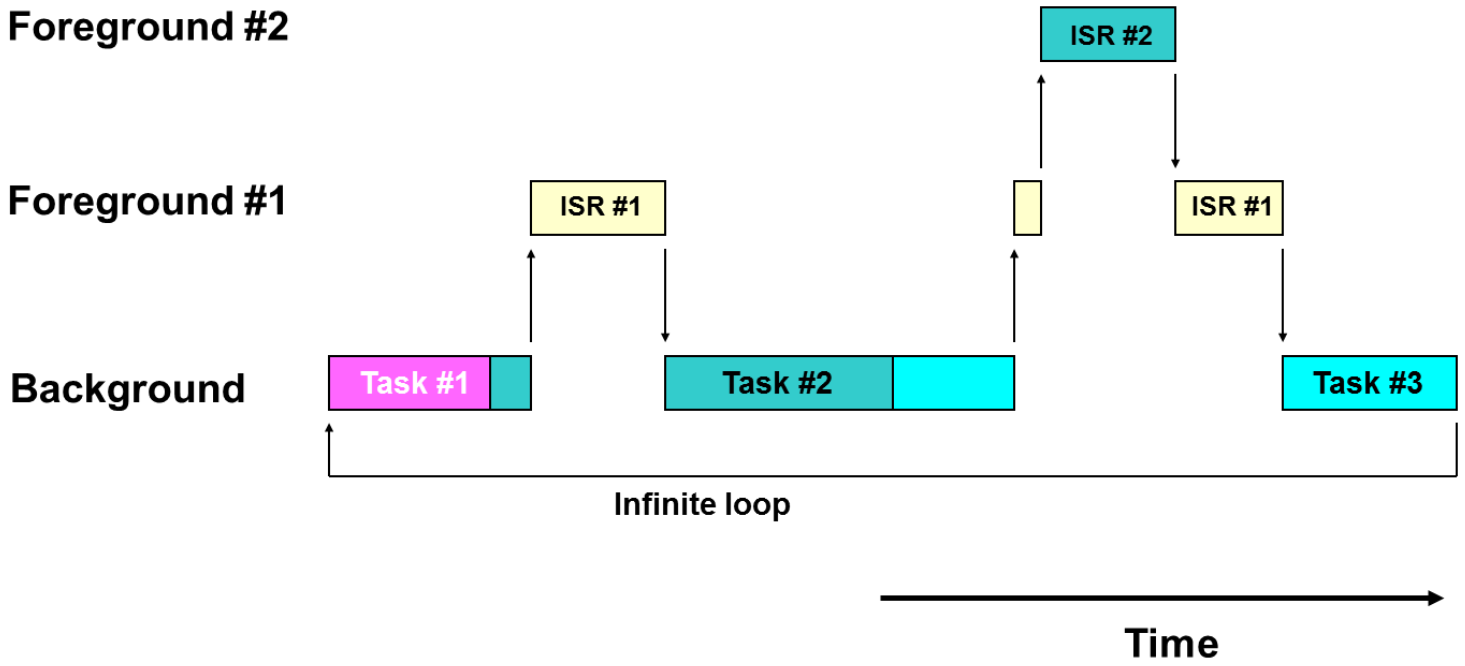
```
void main(void)
{
    /* Initialization */

    while(1)
    {
        /* Input  tasks */

        /* Processing tasks */

        /* Output tasks */
    }
}
```


Foreground/Background




```
int  main (void){

    /* Initialize System */

    while (1) {
        /* Periodic Tasks */
        ADC_Read();
        SPI_Read();
        USB_Packet();
        LCD_Update();
        Audio_Decode();
        File_Write();
    }
}
void  USB_ISR (void){
    Clear interrupt;
    Read packet;
}
```

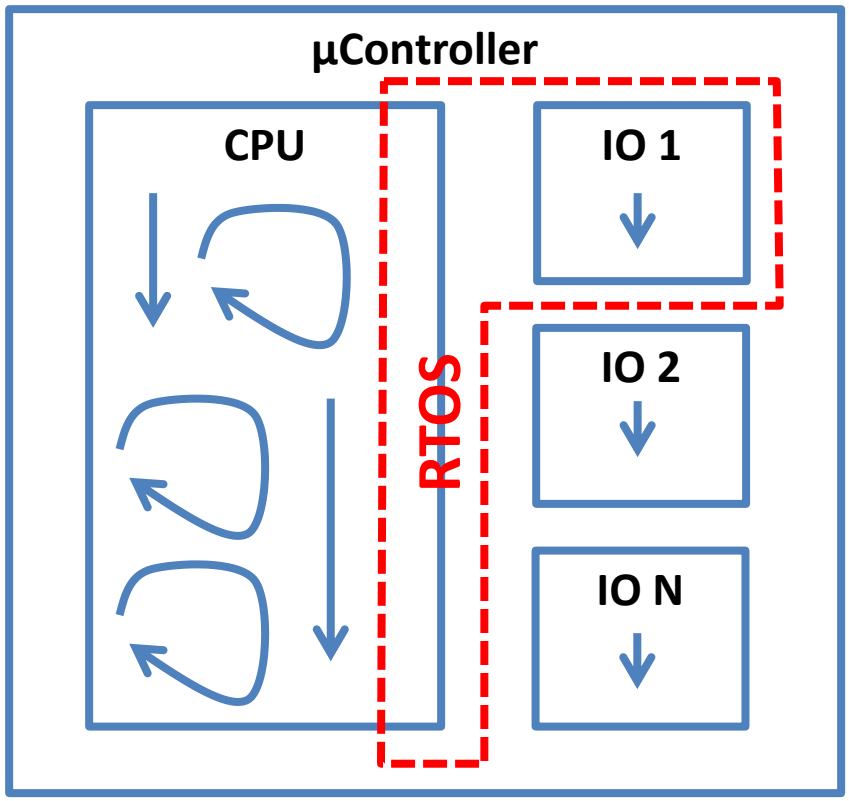

Foreground/Background cont'd



Foreground/Background cont'd

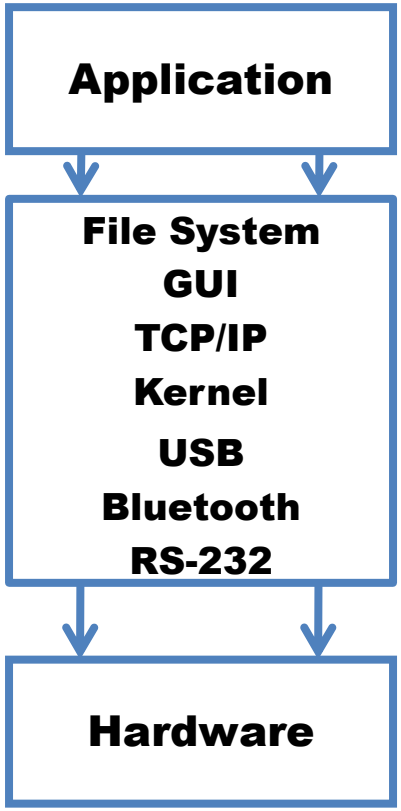
- Pros
  - No upfront cost
  - Minimal training required
  - No need to set aside memory resources to accommodate RTOS
- Cons
  - Difficult to ensure that each operation will meet its deadline
  - High-priority code must be placed in the foreground

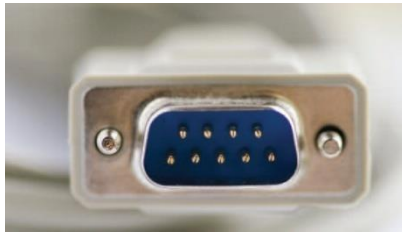
RTOS



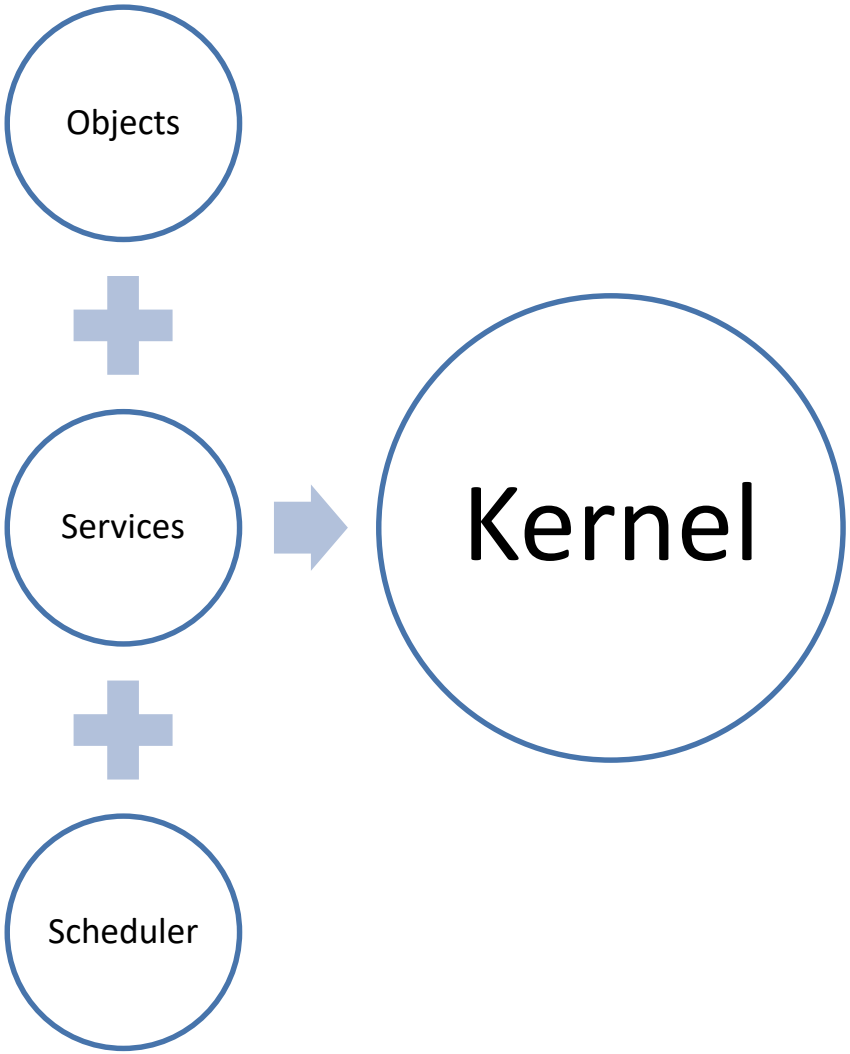
- RT = Correct function @ Correct time
- OS = HW + SW manager
- RTOS = HW + SW manager that can help us ensure having correct function @ correct time

RTOS = Kernel + …

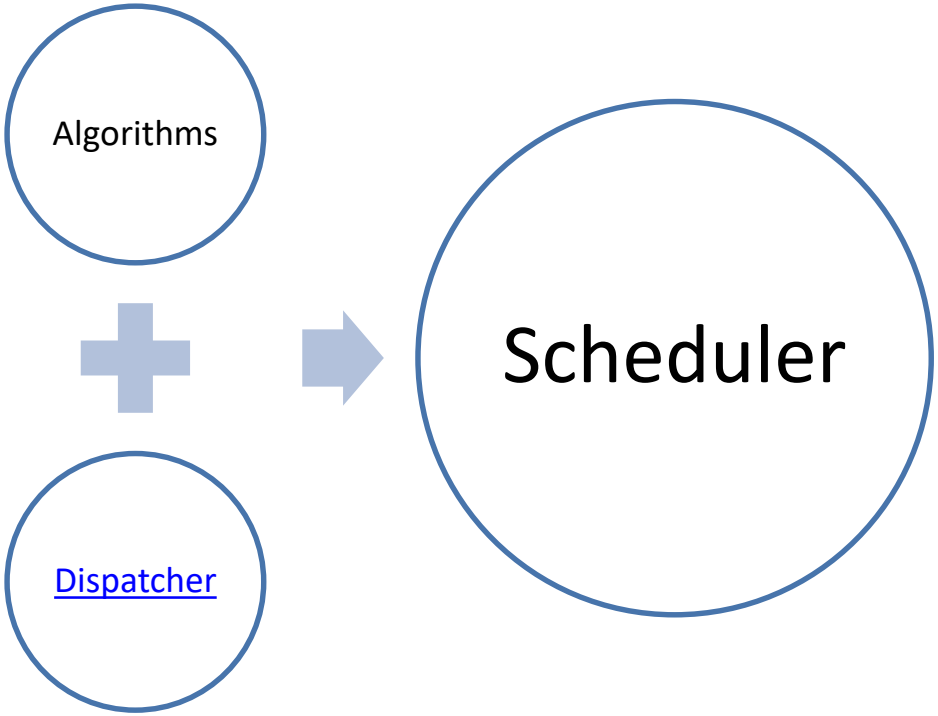

Application
File System
GUI
TCP/IP
Kernel
USB
Bluetooth
RS-232
Hardware



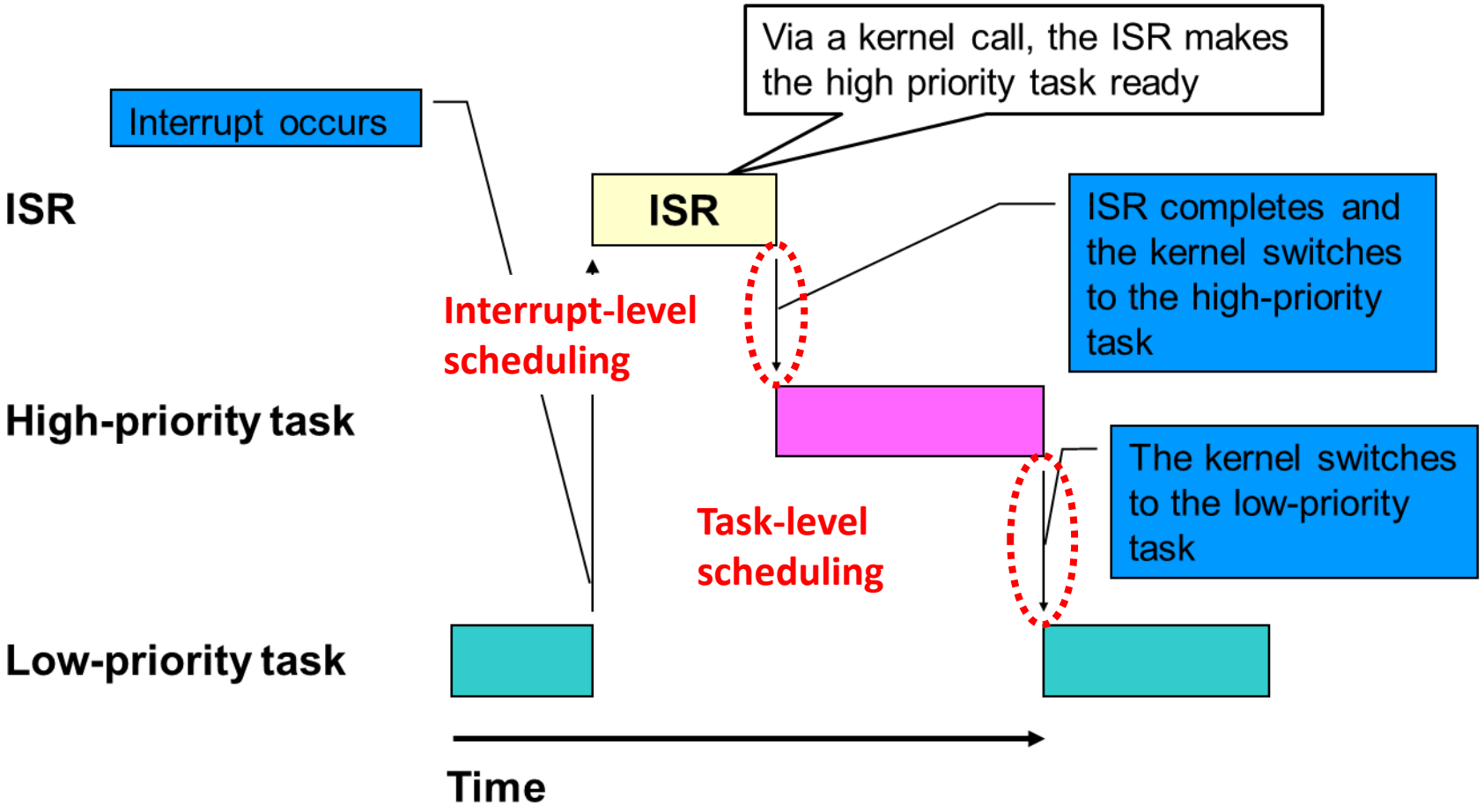
Kernel



Scheduler



RTOS Scheduling



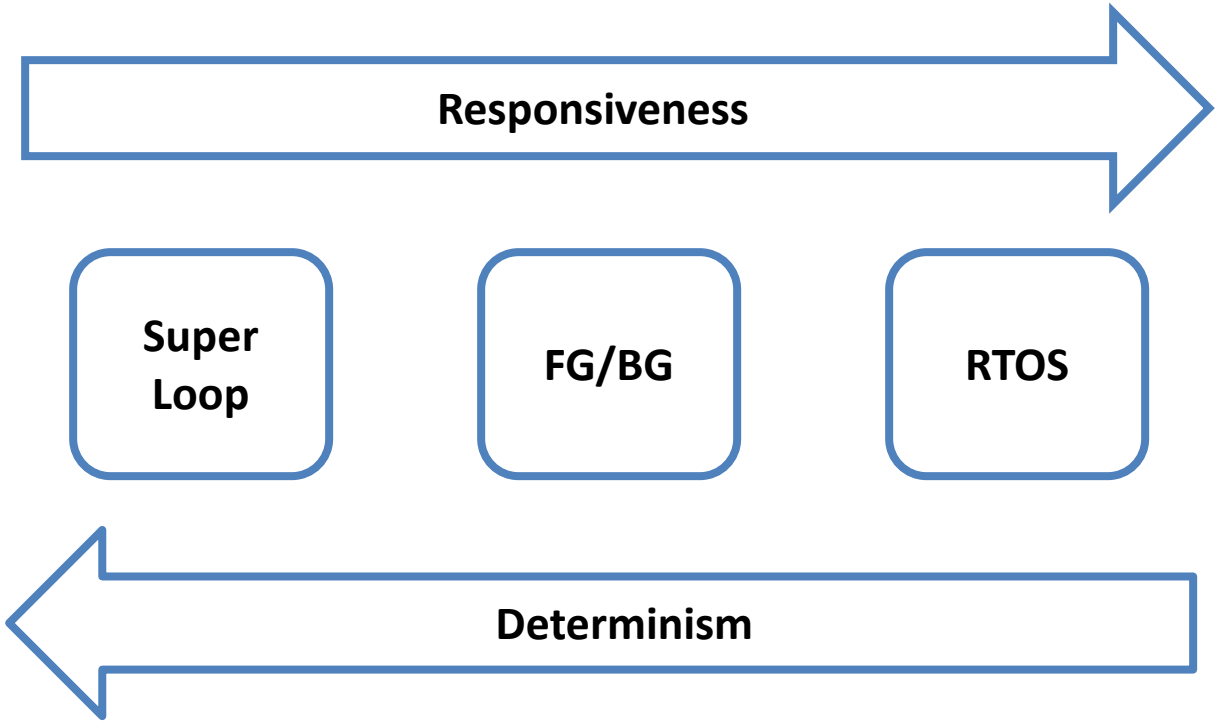
RTOS Benefits

- Developers who use RTOS are freed from implementing a scheduler and related services
- Typically applications that incorporate RTOS are much easier to expand
- The best RTOS have undergone thorough testing

Exercise: Is it Preemptive or not?

	Preemptive	Non-Preemptive
Super Loop		
Foreground/Background		
RTOS		

Conclusion



Outline

- Introduction
- **FreeRTOS Overview**
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

About FreeRTOS

- Market leading RTOS by Real Time Engineers Ltd.
- Professionally developed with strict quality management
- Commercial versions available: OpenRTOS and SafeRTOS
- Documentation available on www.freertos.org and https://www.freertos.org/Documentation/RTOS_book.html
- Free support through its forum

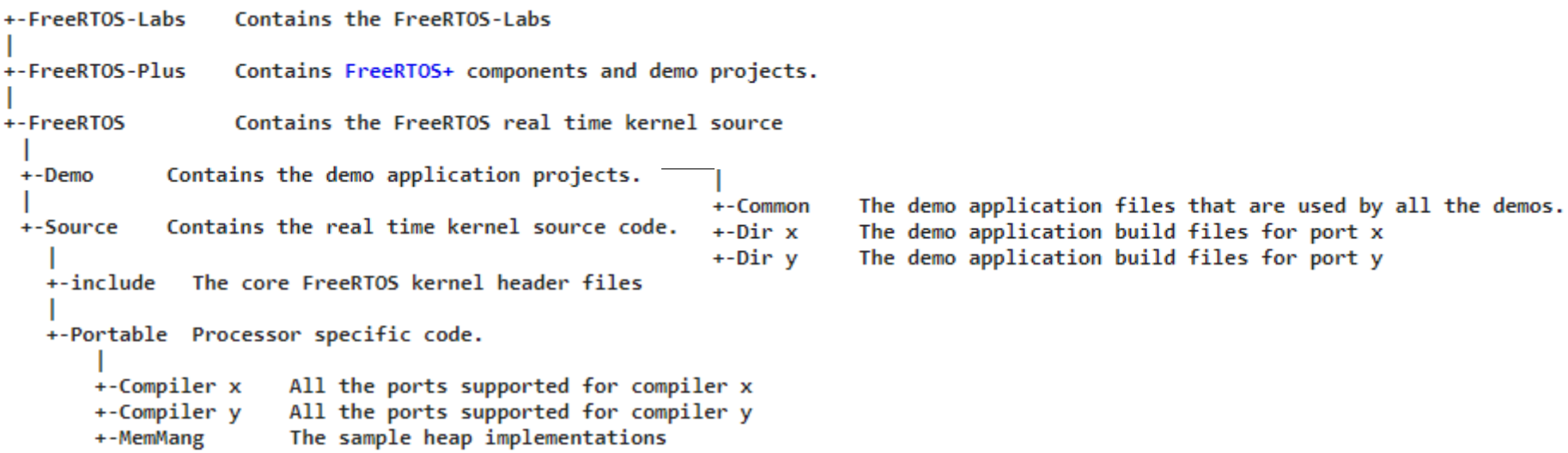
More About FreeRTOS

- MIT License
- Big device support list
- Official and community contributions separated from each other

FreeRTOS Features

- Preemptive or cooperative real-time kernel
- Tiny memory footprint (<10kB ROM and ~0.5kB RAM + task stacks)
- Scalable through configuration
- Ticklessmode for low power applications
- Support synchronization and inter-task communication
- Software timers for tasks scheduling
- Execution trace functionality

FreeRTOS Directory Structure



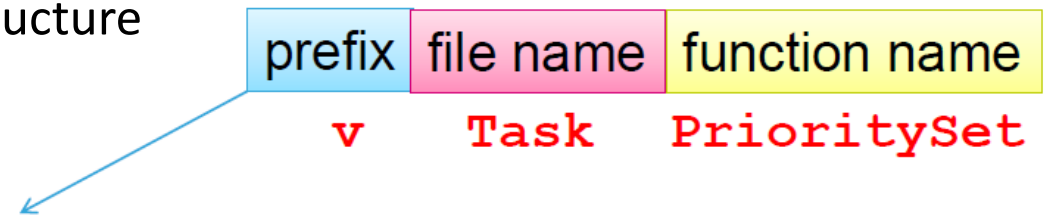
FreeRTOS API Conventions

- Prefixes at variable names

Type	Prefix
uint32_t	ul
uint16_t	us
uint8_t	uc
Non-standard or size_t	x

Type	Prefix
Pointer	p
Enumeration	e

- Functions name structure



v – void
x – returns portBASE_TYPE
prv – private

FreeRTOS API Conventions cont'd

- Prefixes at macros, defines their definition location

Prefix	Location
port	portable.h
task	task.h
pd	projdefs.h
config	FreeRTOSConfig.h
err	projdefs.h

FreeRTOS Project

- It depends on FreeRTOS needed features, target and compiler
- Minimal FreeRTOS needed files are 3 core target-independent files plus one target specific file
- Project can be a new project, an adaptation of an existing project or a demo

Anatomy of a FreeRTOS Project

Item	Description
Source files	FreeRTOS/Source/tasks.c FreeRTOS/Source/queue.c FreeRTOS/Source/list.c FreeRTOS/Source/portable/[compiler]/[architecture]/port.c Any other files in directory of port.c FreeRTOS/Source/portable/MemMang/heap_x.c
Optional source files	FreeRTOS/Source/timers.c FreeRTOS/Source/event_groups.c FreeRTOS/Source/stream_buffer.c FreeRTOS/Source/croutine.c
Header files	FreeRTOS/Source/include FreeRTOS/Source/portable/[compiler]/[architecture] Directory contains FreeRTOSConfig.h

FreeRTOSConfig.h

```
/* Here is a good place to include header files that are required across
your application. */
#include "something.h"

#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configSTACK_DEPTH_TYPE                  uint16_t
#define configMESSAGE_BUFFER_LENGTH_TYPE        size_t

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1
```


FreeRTOSConfig.h cont'd

```
/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0

/* Run time and task stats gathering related definitions. */
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                0
#define configUSE_STATS_FORMATTING_FUNCTIONS    0

/* Co-routine related definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1

/* Software timer related definitions. */
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               3
#define configTIMER_QUEUE_LENGTH                10
#define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE

/* Interrupt nesting behaviour configuration. */
#define configKERNEL_INTERRUPT_PRIORITY         [dependent of processor]
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    [dependent on processor and application]
#define configMAX_API_CALL_INTERRUPT_PRIORITY   [dependent on processor and application]

/* Define to trap errors during development. */
#define configASSERT( ( x ) ) if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )

/* FreeRTOS MPU specific definitions. */
#define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0
```


FreeRTOSConfig.h cont'd

```
/* FreeRTOS MPU specific definitions. */
#define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0

/* Optional functions - most linkers will remove unused functions anyway. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_xResumeFromISR                  1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTaskGetCurrentTaskHandle       1
#define INCLUDE_uxTaskGetStackHighWaterMark     0
#define INCLUDE_xTaskGetIdleTaskHandle          0
#define INCLUDE_eTaskGetState                   0
#define INCLUDE_xEventGroupSetBitFromISR        1
#define INCLUDE_xTimerPendFunctionCall          0
#define INCLUDE_xTaskAbortDelay                 0
#define INCLUDE_xTaskGetHandle                  0
#define INCLUDE_xTaskResumeFromISR              1

/* A header file that defines trace macro can be included here. */

#endif /* FREERTOS_CONFIG_H */
```

- https://www.freertos.org/a00110.html for more details

Starting FreeRTOS, vTaskStartScheduler

- FreeRTOS controls tasks after call
- Create system tasks (idle and optionally timer daemon)
- Return if there is insufficient RTOS heap to system tasks
- Ensure that your target runs this correctly before adding any FreeRTOS features

```
void main(void)
{
    /* Create 1 task @ least here */

    vTaskStartScheduler();
    /* Should I be here? */
}
```

- void vTaskEndScheduler() is not implemented for all ports

Exercise: Development Environment

Outline

- Introduction
- FreeRTOS Overview
- **RTOS Multitasking**
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Task

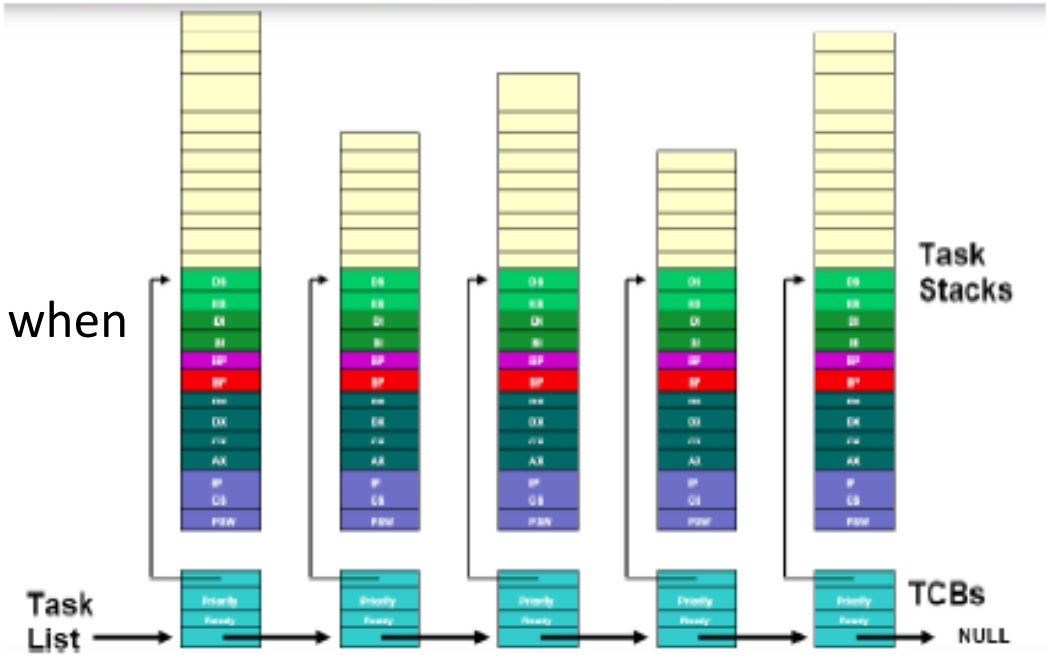
- Task = Function + Context (volatile task state) + Task storage (task stack + TCB)

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared here*/

    /* Task Initialization */


    for( ;; )
    {
        /* Task functionality . */
    }

    /* Tasks are deleted. There are no return to caller */
    vTaskDelete( NULL );
}
```

- Context must be preserved during context switching
- Context = CPU registers + MMU/MPU configuration

Task Control Block

- RTOS uses TCBs for task management
- TCBs reside in RAM
- Every task is assigned a TCB when created



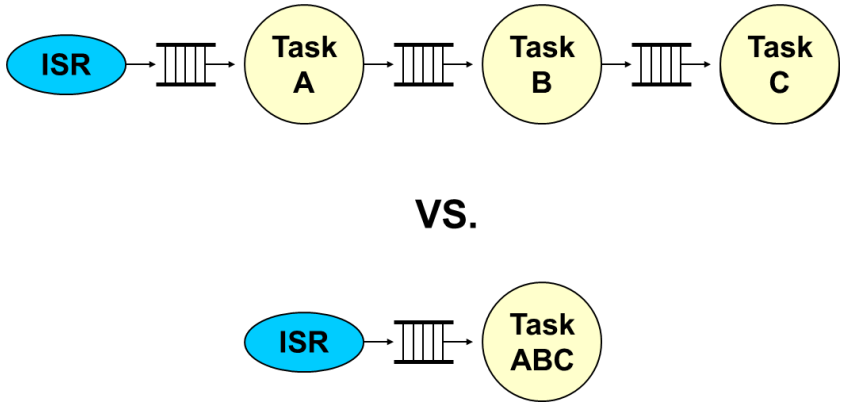
Tasks can be

- System tasks or application tasks
  - System tasks = Created by the RTOS itself for its proper operation (idle task)
  - Application tasks = Created by programmer to implement application

- Run-for-ever tasks or run-to-completion
  - Run-for-ever = They run and never terminate
  - Run-to-completion = The run and will eventually terminate

Identifying Tasks

- Non-trivial
- A poorly partitioned application may fail to meet performance requirements
- Look for activities that can execute in parallel
- Beware of excessive communication and large number of tasks
- Many methods exist



Identifying Tasks: Divide and Conquer

Divide

- Identify code based on the following:
  - IO bound
  - CPU bound
  - Periodicity
- Functions can be either IO or CPU bound
- Functions can be periodic or aperiodic

Conquer (Group)

- Group functions into tasks based on the following:
  - Function cohesion
  - Time cohesion
  - Periodic cohesion
- Function cohesion ➔ Single task or sequential tasks
- Time cohesion ➔ Separate parallel tasks
- Periodic cohesion ➔ Same task w/ counters or different tasks

Identifying Tasks Sanity Check

- For each task regardless of the execution schedule:
  - T = WCET
  - D = Deadline
  - P = Period

$$T \leq D \leq P$$

$$T = D = P$$

$$T = D \leq P$$

Identifying Task Priorities

- If priorities are assigned arbitrarily, the benefits of using RTOS may not be realized
- When multiple tasks have important deadlines, assigning priorities can be particularly difficult
- Many methods exist

Identifying Task Priorities: Gomaa Criterion

- Tasks can be classified according to 2 attributes:
  - Criticality
  - Urgency

Urgent	Critical	Priority
No	No	Lowest
No	Yes	Lower
Yes	No	Higher
Yes	Yes	Highest

Identifying Task Priorities: RMS

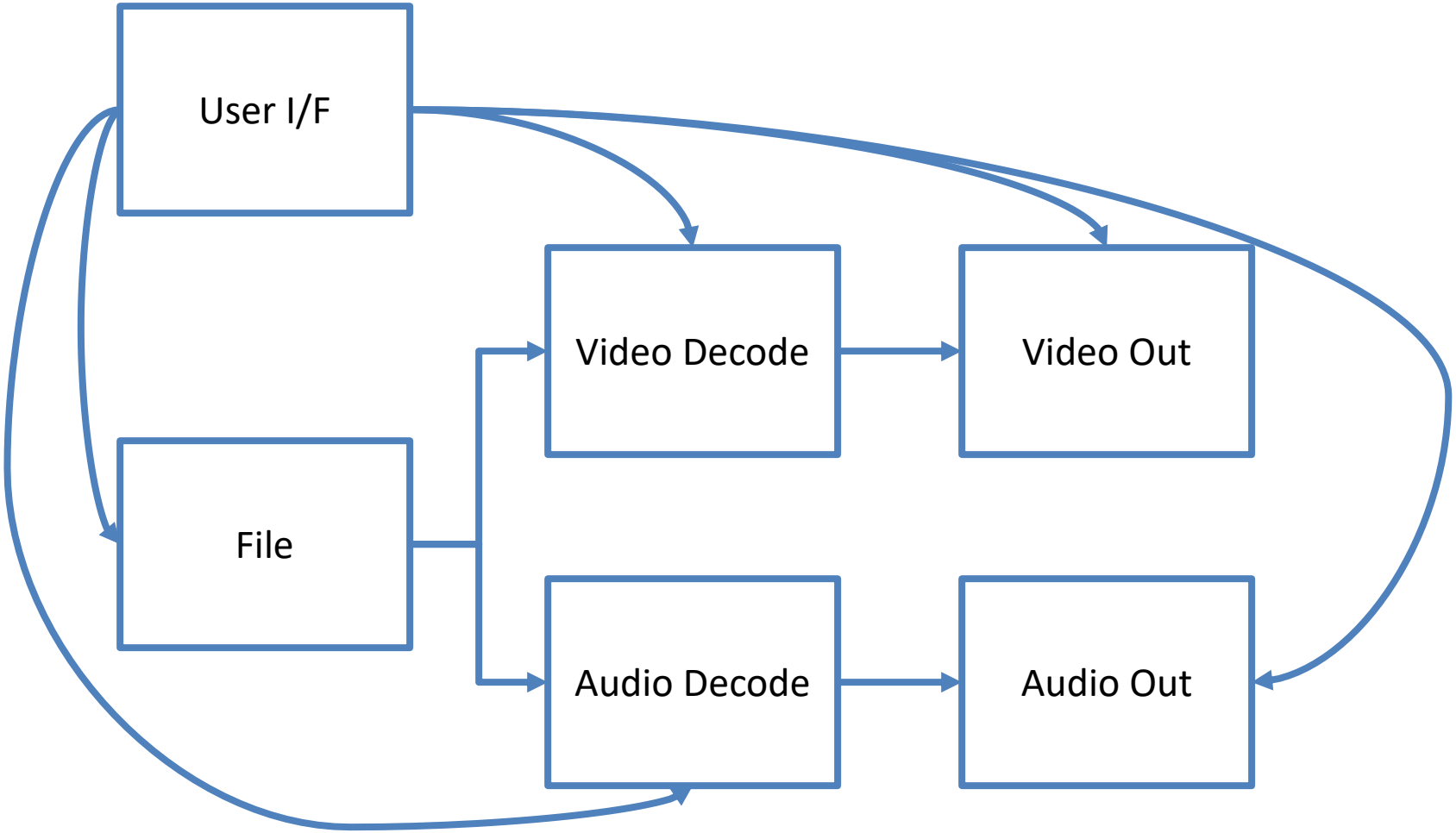
- Special case of Gomaa criterion 
- The tasks with the highest frequencies are given the highest priorities
- Optimal
- Can be applied iff:
  - Each task runs periodically
  - A given task always completes its work within a fixed amount of time
  - Tasks do not interact

Assigning Priorities Sanity Check

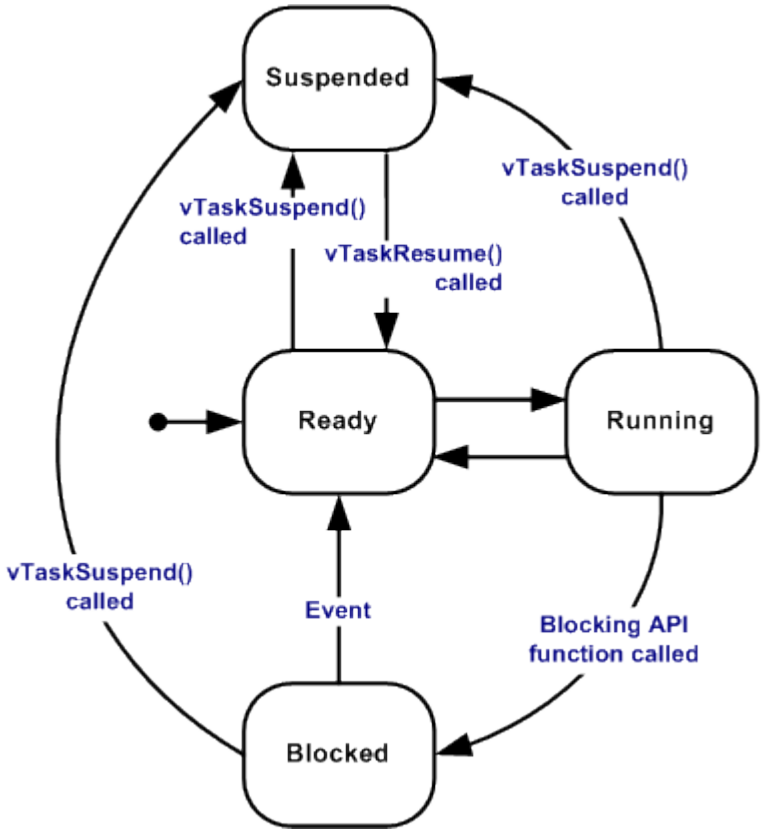
- For each task considering the execution schedule:
  - T = WCET
  - B = Total blockage time within the period
  - D = Deadline
  - P = Period

$$T + B \leq D = P$$

Example: Video Player

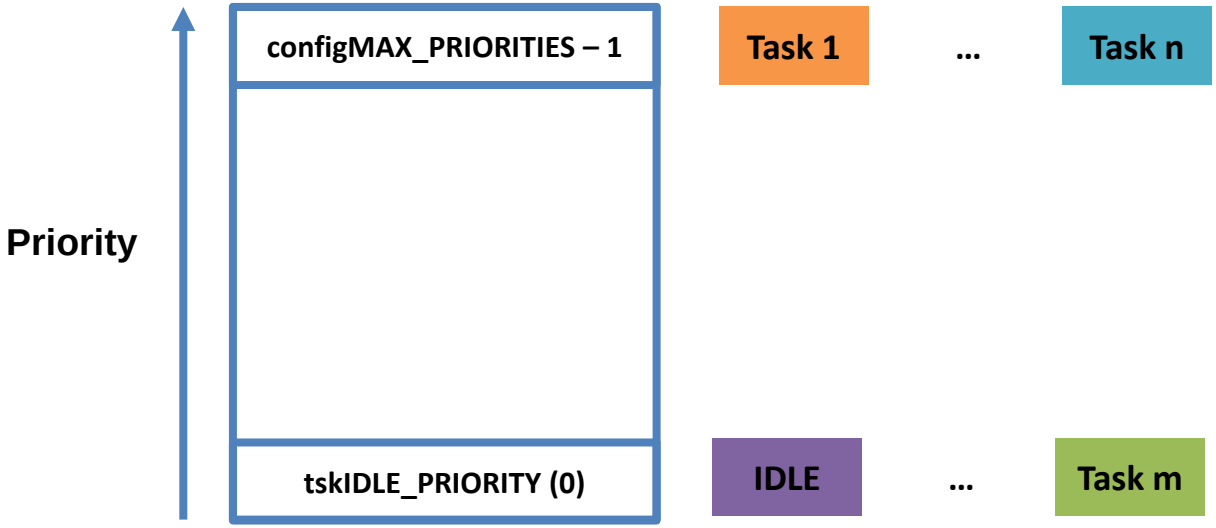


Task States


Suspended
vTaskSuspend()
called
vTaskResume()
called
vTaskSuspend()
called
Ready
Running
vTaskSuspend()
called
Event
Blocking API
function called
Blocked

Task Priority

- Keep configMAX_PRIORITIES as minimum as possible.
- configUSE_PORT_OPTIMISED_TASK_SELECTION can limit configMAX_PRIORITIES
  - If 0 or undefined, no limit (more RAM and affects WCET)
  - If 1, limited to 32 (fixed RAM and does not affect WCET)
    - In some architectures only



Task Scheduling

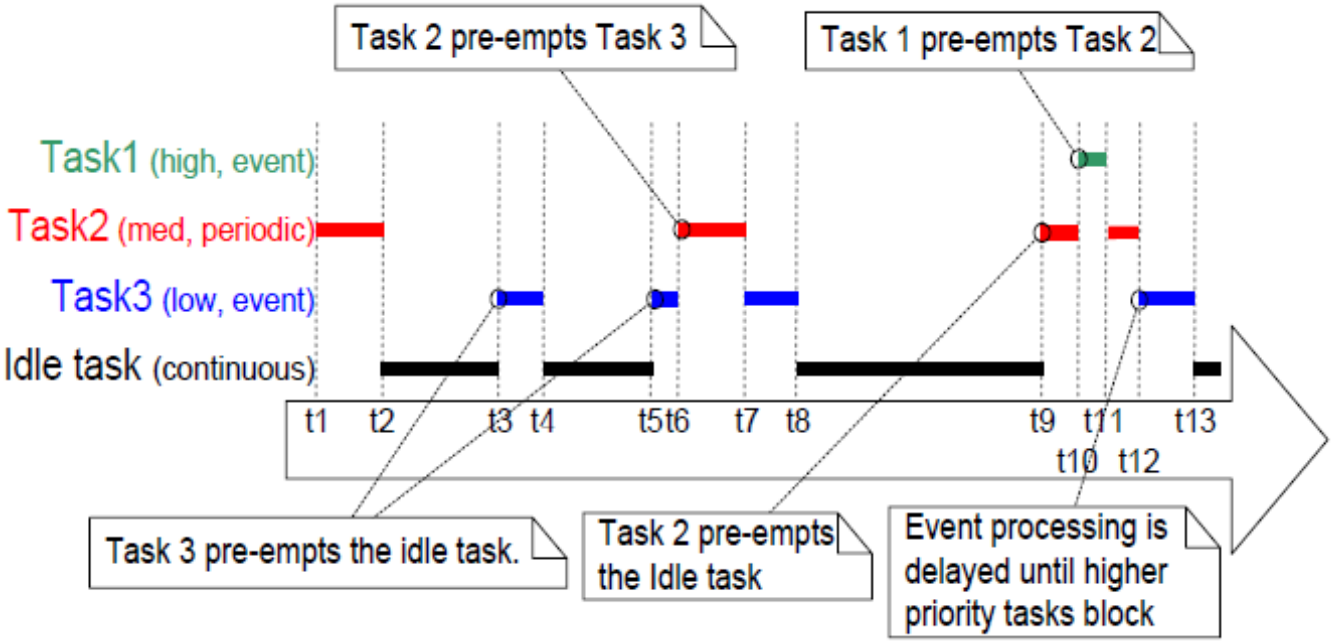
Scheduling Algorithm	configUSE_PREEMPTION	configUSE_TIME_SLICING
Fixed Priority Preemptive Scheduling w/ Time Slicing	1	1
Fixed Priority Preemptive Scheduling w/o Time Slicing	1	0
Cooperative Scheduling	0	X

Fixed Priority Preemptive Scheduling w/ Time Slicing

- Tasks are preempted with higher priority tasks.
- Tasks of same priority share CPU equally based on time slicing.
- Context switch when running task blocks, suspends, yields the CPU (taskYIELD()), its time slice expire or when a higher priority task is ready
- Time slice starts on each tick interrupt.
  - configTICK_RATE_HZ

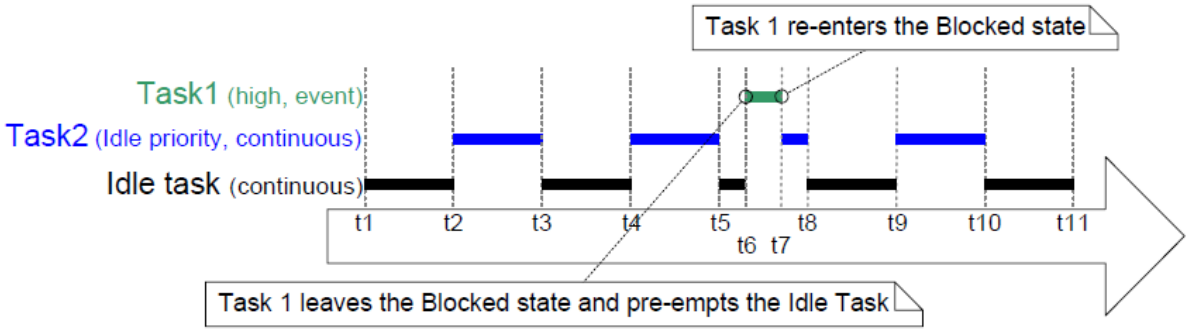
Fixed Priority Preemptive Scheduling w/ Time Slicing – Unique Priorities

- Unique priorities

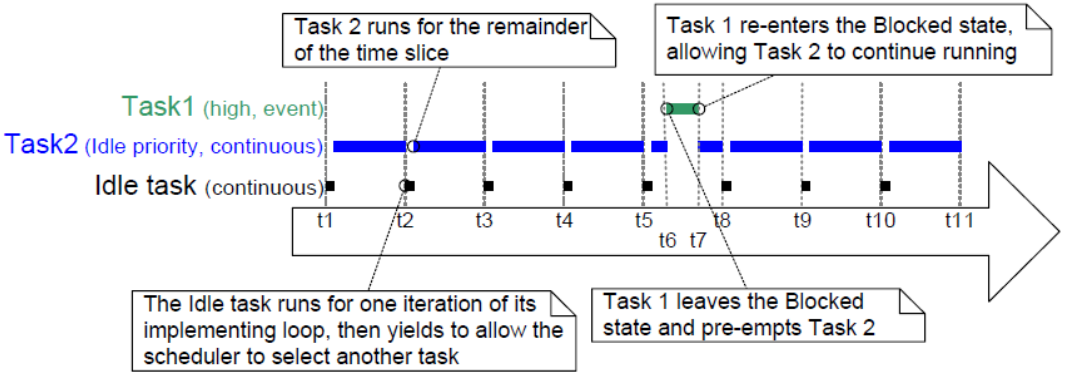


Fixed Priority Preemptive Scheduling w/ Time Slicing – Shared Priorities

- configIDLE_SHOULD_YIELD = 0

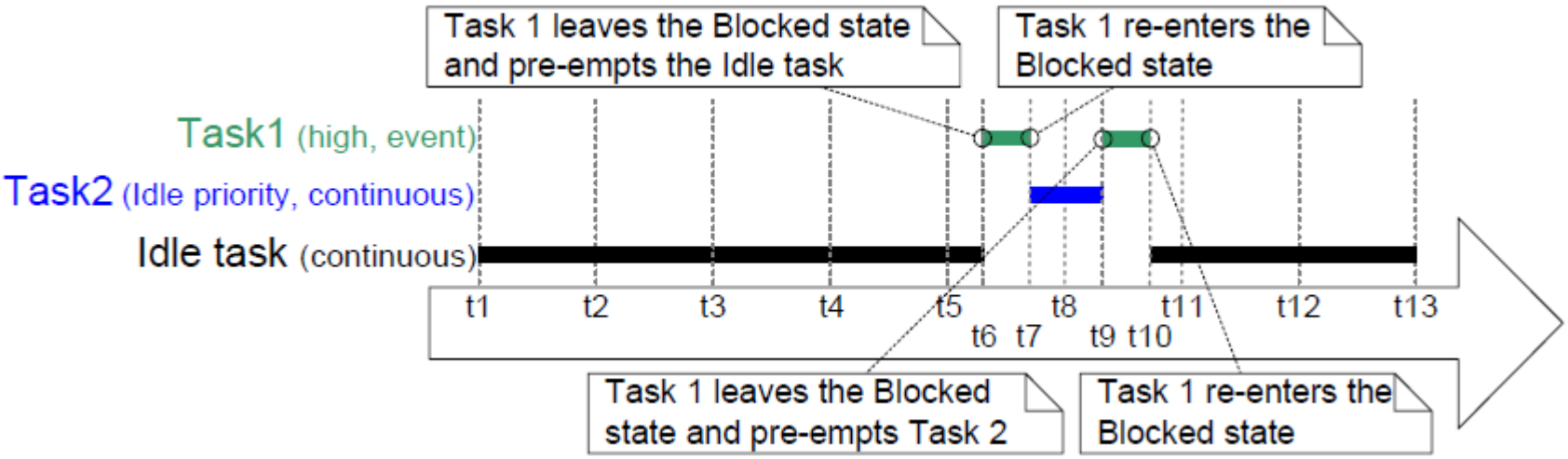


- configIDLE_SHOULD_YIELD = 1



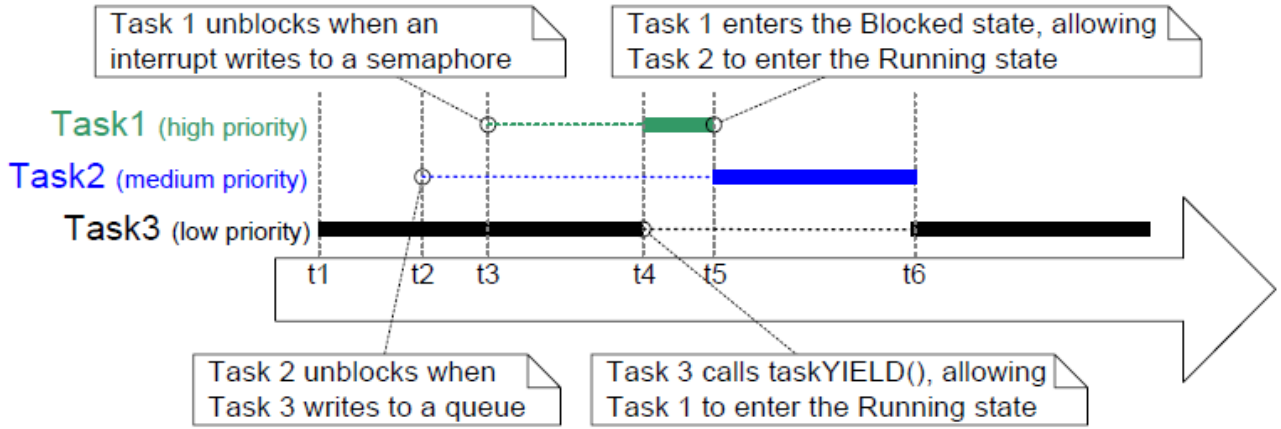
Fixed Priority Preemptive Scheduling w/o Time Slicing

- Do it w/ care



Cooperative Scheduling

- Tasks are not preempted with higher priority tasks.
- Context switch when running task blocks, suspends or yields the CPU (taskYIELD())
- No time slice preemption



Idle Task and Idle Task Hook

Idle Task

- Created when the scheduler starts
- Free memory allocated by FreeRTOS to tasks that have been deleted
  - Make sure it does not starve is you use vTaskDelete().
- Application tasks can share Idle task priority

Idle Task Hook

- Called every cycle from Idle Task
- Extends Idle task functionality
- Enabled by configUSE_IDLE_HOOK
- Should not call a blocking API
- Should return to caller if Idle tasks is freeing memory
- What is the alternative? Can you compare it to the alternative?

Example: Idle Task Hook

```c
/* Idle hook function */
void vApplicationIdleHook(void)
{
    ulIdleCycleCount++;
}
```

```c
/* Task printing Idle Hook data */
void vTaskFunction(void *pvParameters)
{
    char *pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS(250);

    pcTaskName = (char *) pvParameters;

    for(;;)
    {
        vPrintStringAndNumber(pcTaskName, ulIdleCycleCount);
        vTaskDelay(xDelay250ms);
    }
}
```


FreeRTOS Task APIs

Creation	Control	Utilities
xTaskCreate xTaskCreateStatic vTaskDelete	vTaskDelay vTaskDelayUntil uxTaskPriorityGet vTaskPrioritySet vTaskSuspend vTaskResume xTaskAbortDelay	uxTaskGetSystemState vTaskGetInfo xTaskGetApplicationTaskTag xTaskGetCurrentTaskHandle xTaskGetHandle xTaskGetIdleTaskHandle uxTaskGetStackHighWaterMark eTaskGetState pcTaskGetName xTaskGetTickCount xTaskGetSchedulerState xTaskGetNumberOfTasks vTaskList vTaskGetRunTimeStats vTaskGetIdleRunTimeCounter vTaskSetApplicationTaskTag xTaskCallApplicationTaskHook vTaskSetThreadLocalStoragePointer vTaskGetThreadLocalStoragePointer vTaskSetTimeOutState xTaskCheckForTimeOut

Creating a Task, xTaskCreate

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char * const pcName,
                       configSTACK_DEPTH_TYPE usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *pxCreatedTask
                      );
```

- pcName length ≤ configMAX_TASK_NAME_LEN
- configMINIMAL_STACK_SIZE ≤ usStackDepth ≤ size_t/stack width
- Required stack and TCB are automatically allocated from FreeRTOS heap
  - configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- If uxPriority > configMAX_PRIORITIES – 1, it will be capped silently
- Can create a privileged task (if we have MPU) by setting bit portPRIVILEGE_BIT in uxPrriority (2|portPRIVILEGE_BIT)
- Return pdPASS or errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

Creating a Task, xTaskCreateStatic

```
TaskHandle_t xTaskCreateStatic(TaskFunction_t pxTaskCode,
                               const char * const pcName,
                               const uint32_t ulStackDepth,
                               void * const pvParameters,
                               UBaseType_t uxPriority,
                               StackType_t * const puxStackBuffer,
                               StaticTask_t * const pxTaskBuffer);
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- puxStackBuffer is used as a task stack with size ≥ ulStackDepth
- pxTaskBuffer is used as task TCB
- Return task's handle or NULL

Deleting a Task, vTaskDelete

```
void vTaskDelete(TaskHandle_t xTask);
```

- INCLUDE_vTaskDelete must be 1
- If xTask is NULL, the task deletes itself

Delaying a Task, vTaskDelay/vTaskDelayUntil

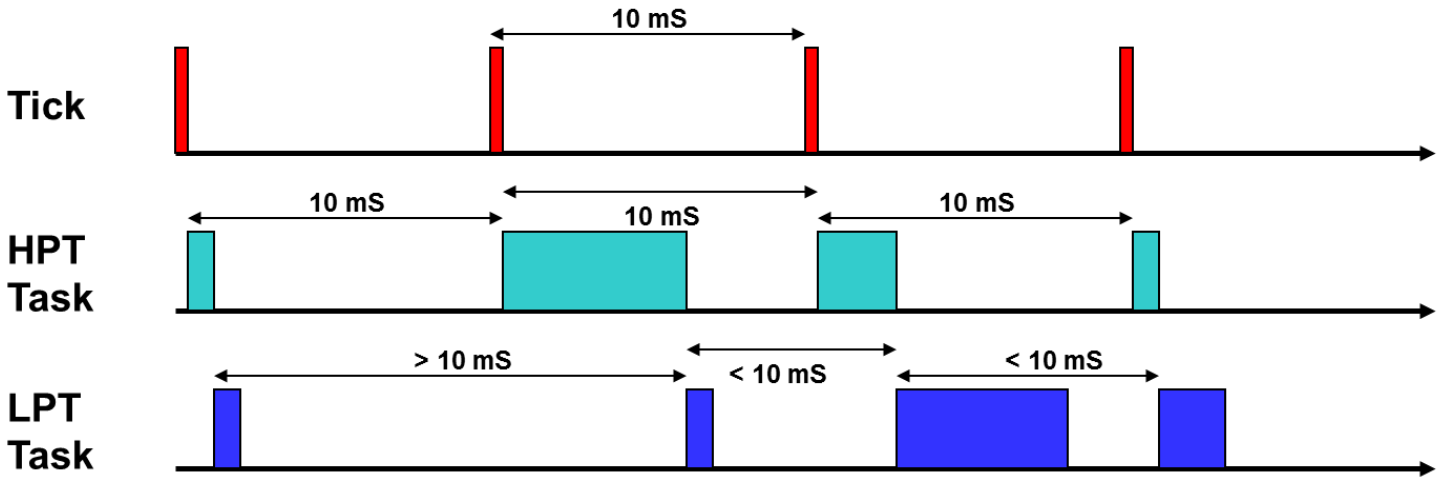
```
void vTaskDelay(const TickType_t xTicksToDelay);
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime,
                     const TickType_t xTimeIncrement);
```

- INCLUDE_vTaskDelay/INCLUDE_vTaskDelayUntil must be 1
- vTaskDelay() specifies a time at which the task wishes to unblock relative to its call
- Use pdMS_TO_TICKS to calculate real time from the tick rate
- vTaskDelayUntil() = vTaskDelay() but w/ absolute reference
- vTaskDelayUntil() is used by periodic tasks to ensure a constant execution period = xTimeIncrement
- Re-calculate xTimeIncrement if periodic execution is halted for any reason
- Task unblock time = *pxPreviousWakeTime + xTimeIncrement

Delaying a Task, vTaskDelay/vTaskDelayUntil cont'd

```
// Perform an action every 10 ticks.
void vTaskFunction(void * pvParameters)
{
TickType_t xLastWakeTime;
const TickType_t xFrequency = 10;

 xLastWakeTime = xTaskGetTickCount();

 for( ;; )
      {
          // Wait for the next cycle.
          vTaskDelayUntil(&xLastWakeTime, xFrequency);

          // Perform action here.
      }
}
```


Be Aware of Jitter



Getting/Setting a Task Priority, uxTaskPriorityGet/vTaskPrioritySet

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t xTask );
void vTaskPrioritySet(TaskHandle_t xTask,
                      UBaseType_t uxNewPriority );
```

- INCLUDE_uxTaskPriorityGet/INCLUDE_vTaskPrioritySet must be 1
- If xTask is NULL, the task gets/sets own priority

Suspending/Resuming a Task, vTaskSuspend/vTaskResume

```
void vTaskSuspend(TaskHandle_t xTaskToSuspend);
void vTaskResume(TaskHandle t xTaskToResume);
```

- INCLUDE_vTaskSuspend must be 1
- vTaskSuspend() is not accumulative and task can be resumed by a single vTaskResume()
- If xTaskToSuspend is NULL, the task suspends itself

Aborting a Task Delay, xTaskAbortDelay

```
BaseType_t xTaskAbortDelay(TaskHandle_t xTask);
```

- INCLUDE_xTaskAbortDelay must be 1
- Return pdPASS or pdFAIL

Getting a Task Handle, xTaskGetXXXHandle

```
TaskHandle_t xTaskGetHandle(const char *pcNameToQuery);
TaskHandle_t xTaskGetCurrentTaskHandle(void);
TaskHandle_t xTaskGetIdleTaskHandle(void);
```

- INCLUDE_xTaskGetHandle/INCLUDE_xTaskGetCurrentTaskHandle/ INCLUDE_xTaskGetIdleTaskHandle must be 1
- xTaskGetHandle() is slow, should be used once per task
- xTaskGetHandle() return handle if found or NULL

Getting a Task Info, vTaskGetInfo/ uxTaskGetSystemState

```
void vTaskGetInfo(TaskHandle_t xTask, TaskStatus_t *pxTaskStatus,
    BaseType_t xGetFreeStackSpace, eTaskState eState );
UBaseType_t uxTaskGetSystemState(TaskStatus_t * const pxTaskStatusArray,
   const UBaseType_t uxArraySize, unsigned long * const pulTotalRunTime);
```

- Used for debugging only
- configUSE_TRACE_FACILITY must be 1
- They are time consuming and suspend scheduling
- If xGetFreeStackSpace is pdFALSE, stack high watermark checking will be skipped in TaskStatus_t
- If eState is eInvalid, task state will not be skipped in TaskStatus
- pulTotalRunTime is set to total run time since booting if configGENERATE_RUN_TIME_STATS is 1

Tagging a Task, vTaskSetApplicationTaskTag/ xTaskGetApplicationTaskTag

```
void vTaskSetApplicationTaskTag(TaskHandle_t xTask,
    TaskHookFunction_t pxTagValue);
TaskHookFunction_t xTaskGetApplicationTaskTag(TaskHandle_t xTask);
```

- configUSE_APPLICATION_TASK_TAG must be 1
- A tag can be any value and understood only by application writer
- A tag can be used to define a task hook
- If xTask is NULL, the task gets/sets own tag
- TaskHookFunction_t is a pointer to function that take a void * parameter, and return BaseType_t

Calling a Task Hook, xTaskCallApplicationTaskHook

```
BaseType_t xTaskCallApplicationTaskHook(TaskHandle_t xTask,
        void *pvParameter);
```

- configUSE_APPLICATION_TASK_TAG must be 1
- If xTask is NULL, the task calls own hook

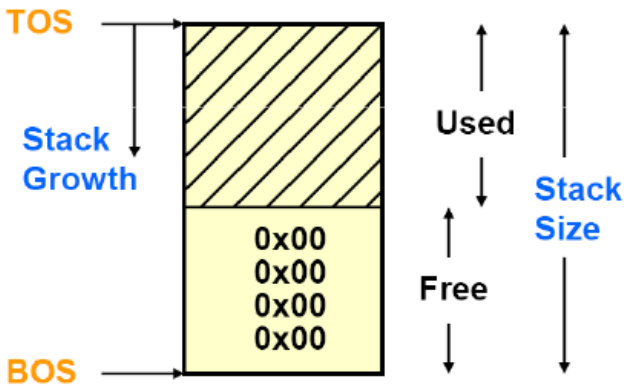
Calling a Task Hook, xTaskCallApplicationTaskHook cont'd

```
static BaseType_t prvExampleTaskHook(void * pvParameter)
{
    /* Perform some action */
    return 0;
}


void vAnotherTask(void *pvParameters)
{
    /* Register our callback function. */
    vTaskSetApplicationTaskTag(NULL, prvExampleTaskHook);

    for( ;; )
    {
        /* Rest of task code goes here. */
    }
}

/* Calling hook @ context switch */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook(pxCurrentTCB, 0)
```


Checking a Task Stack, uxTaskGetStackHighWaterMark

```
UBaseType_t uxTaskGetStackHighWaterMark(TaskHandle_t xTask);
configSTACK_DEPTH_TYPE uxTaskGetStackHighWaterMark2(TaskHandle_t xTask);
```

- INCLUDE_uxTaskGetStackHighWaterMark must be 1
- uxTaskGetStackHighWaterMark2() returns a user definable type
- If xTask is NULL, the task checks own hook



Other Task Getters

```
eTaskState eTaskGetState(TaskHandle_t xTask);
```

- INCLUDE_eTaskGetState must be 1
- Task states are eReady, eRunning, eBlocked, eSuspended or eDeleted

```
char * pcTaskGetName(TaskHandle_t xTaskToQuery);
```

- If xTaskToQuery is NULL, the task checks own name

```
volatile TickType_t xTaskGetTickCount(void);
```


Other Task Getters cont'd

```
BaseType_t xTaskGetSchedulerState(void);
```

- INCLUDE_xTaskGetSchedulerState or configUSE_TIMERS must be 1
- Return taskSCHEDULER_NOT_STARTED, taskSCHEDULER_RUNNING or taskSCHEDULER_SUSPENDED

```
UBaseType_t uxTaskGetNumberOfTasks(void);
```

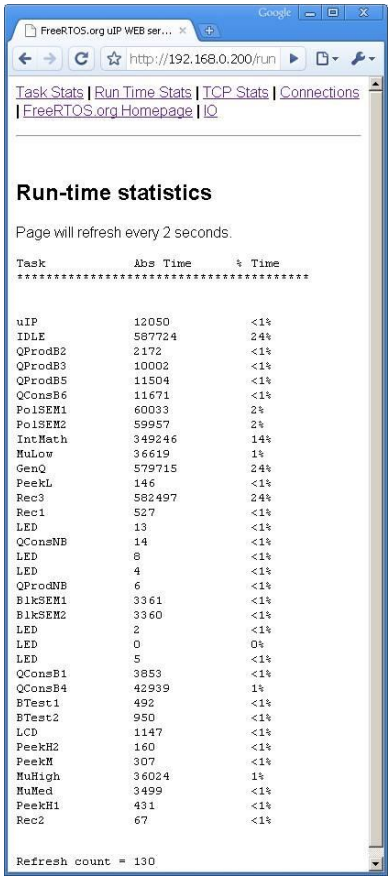
- Running, ready, blocked, suspended and deleted but not yet freed tasks

Task Run-Time Statistics

- FreeRTOS can optionally collect processing time info per task
  - Absolute time
  - %
- Up to the user to select suitable time base (@ least 10 times faster than tick) for their app
- Configuration and Usage
  - configGENERATE_RUN_TIME_STATS must be 1
  - Define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() to configure statistics timer with suitable time base
  - Define portGET_RUN_TIME_COUNTER_VALUE() to return current timer counter

Example: Task Run-Time Statistics

```
/* Defined in main.c. */
void vConfigureTimerForRunTimeStats(void)
{
    /* Power up and feed the timer with a clock. */
    /* Reset Timer */
    /* Just count up. */
    /* Set Prescale to a good time-base */
    /* Start the timer. */
}

/* Defined in FreeRTOSConfig.h. */
extern void vConfigureTimerForRunTimeStats(void);
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() \
                vConfigureTimerForRunTimeStats()
#define portGET_RUN_TIME_COUNTER_VALUE() /* Get timer counter */
```



Getting Run-Time Statistics, vTaskGetRunTimeStats/ vTaskGetIdleRunTimeCounter

```
void vTaskGetRunTimeStats(char *pcWriteBuffer);
```

- Used for debugging only and disable interrupts while running
- configUSE_STATS_FORMATTING_FUNCTIONS must be 1
- Calls uxTaskGetSystemState(), then formats its raw data into a human readable table
- pcWriteBuffer should be large enough (~40 bytes/task)

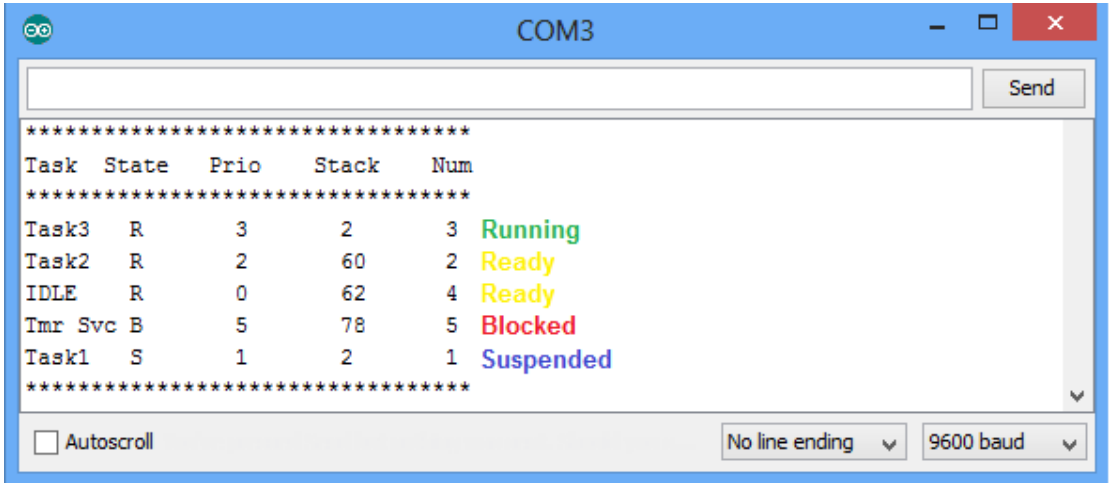
```
TickType_t xTaskGetIdleRunTimeCounter(void);
```

- INCLUDE_xTaskGetIdleTaskHandle must be 1

Listing Tasks in System, vTaskList

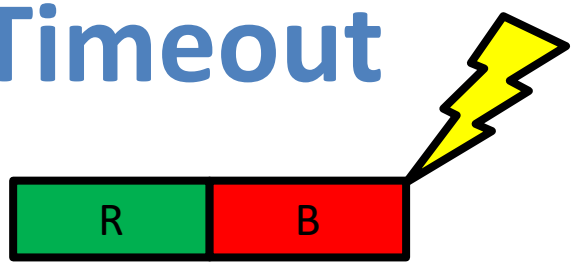
```
void vTaskList(char *pcWriteBuffer);
```

- Used for debugging only and disable interrupts while running
- configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must be 1
- Calls uxTaskGetSystemState(), then formats its raw data into a human readable table
- pcWriteBuffer should be large enough (~40 bytes/task)

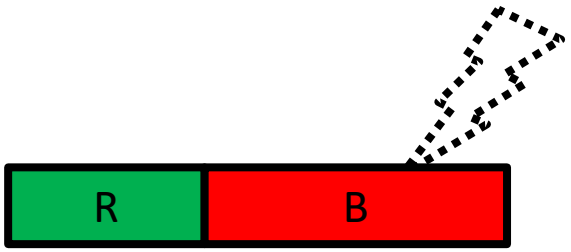


Task Multiple Block Timeout

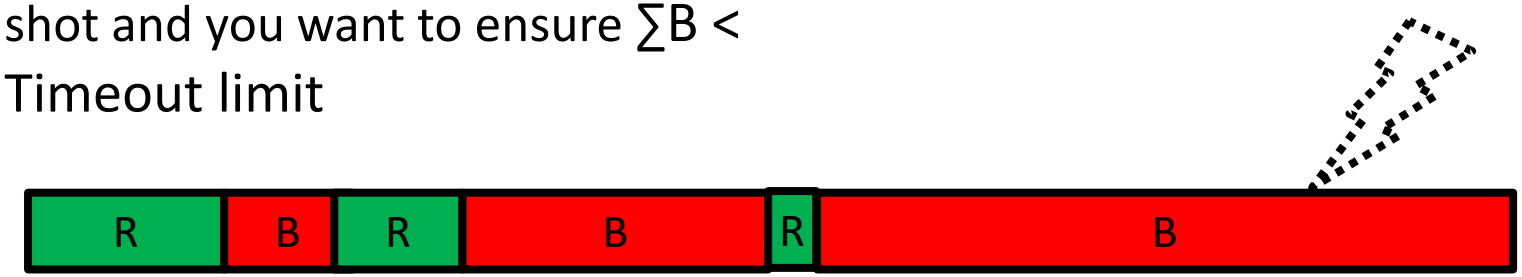
- Task blocks for event w/ timeout but event occurs before timeout



- Task blocks for event w/ timeout and times out



- What if the block time is not one shot and you want to ensure ∑B < Timeout limit



Checking a Task Multiple Block Timeout, vTaskSetTimeOutState/xTaskCheckForTimeOut

```
void vTaskSetTimeOutState(TimeOut_t * const pxTimeOut);
BaseType_t xTaskCheckForTimeOut(TimeOut_t * const pxTimeOut,
        TickType_t * const pxTicksToWait);
```

- These are for advanced use only
- vTaskSetTimeOutState() sets the initial condition
- pxTimeOut will be initialized to hold information necessary to determine if a timeout has occurred
- xTaskCheckForTimeOut() checks for a timeout condition and adjust the remaining block time if a timeout has not occurred
- pxTicksToWait is used to pass out an adjusted remaining block time
- xTaskCheckForTimeOut return pdTRUE if no block time remains, pdFALSE otherwise

Example: Task Timeout

```
/* Receive uxWantedBytes from an Rx buffer that is filled by a UART interrupt */
size_t xUART_Receive(uint8_t *pucBuffer, size_t uxWantedBytes)
{
    size_t uxReceived = 0;
    TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
    TimeOut_t xTimeOut;
    /* Initialize xTimeOut w/ time @ which this function called . */
    vTaskSetTimeOutState( &xTimeOut );
    /* Loop until buffer contains wanted number of bytes, or timeout occurs. */
    while( UART_bytes_in_rx_buffer(pxUARTInstance) < uxWantedBytes )
    {
        /* Adjusting xTicksToWait to account for time spent so far in Blocked state . */
        if( xTaskCheckForTimeOut(&xTimeOut, &xTicksToWait) != pdFALSE )
        {
            /* Timed out before the wanted number of bytes */
            break;
        }
        /* Wait for a maximum of xTicksToWait ticks to be notified that more
        data placed into the buffer. */
        ulTaskNotifyTake(pdTRUE, xTicksToWait);
    }
    /* Read uxWantedBytes from buffer pucBuffer. Actual number of bytes
    read, might be < uxWantedBytes is returned. */
    uxReceived = UART_read_from_receive_buffer(pxUARTInstance, pucBuffer, uxWantedBytes);
    return uxReceived;
}
```


Thread Local Storage

- Allows application writer to store values inside a task's TCB
- Array of pointers with size equal to configNUM_THREAD_LOCAL_STORAGE_POINTERS

```
void vTaskSetThreadLocalStoragePointer(TaskHandle_t xTaskToSet,
        BaseType_t xIndex, void *pvValue );
void *pvTaskGetThreadLocalStoragePointer(TaskHandle_t xTaskToQuery,
        BaseType_t xIndex );
```

- If xTastToSetxTaskToQuery is NULL, the task sets/gets own TLS
- xIndex must be < configNUM_THREAD_LOCAL_STORAGE_POINTERS

Exercise: RTOS Multitasking

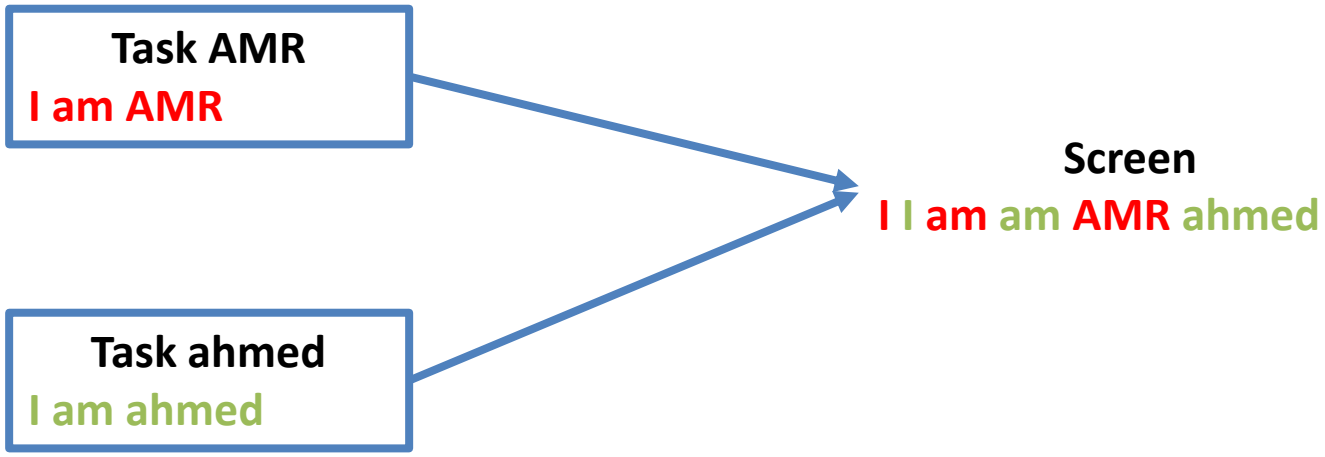
Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- **Inter-task Access Synchronization**
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Shared Resources

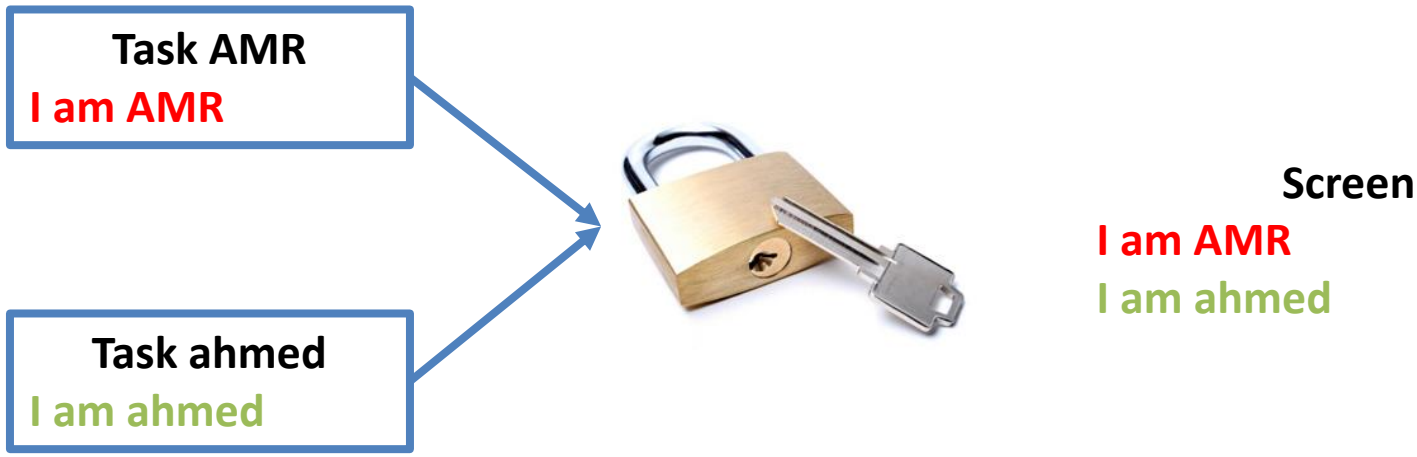
- Are used by multiple contexts
  - A global variable/data structure
  - Peripheral devices
  - Non thread-safe libraries

- Race condition, major problem in multi-context execution



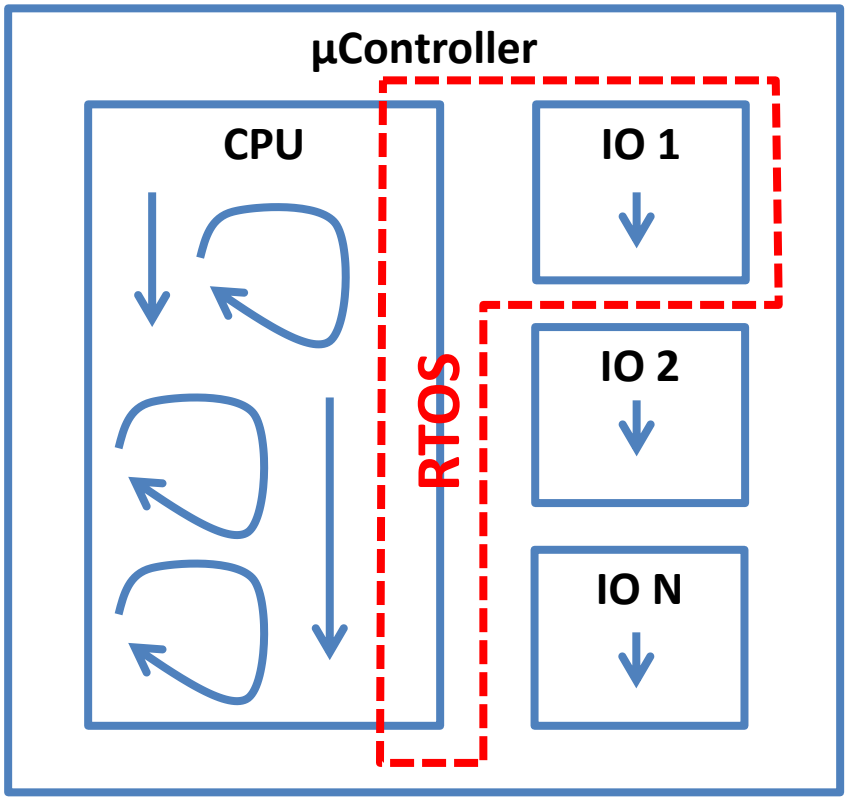
Protecting Shared Resources

- Access to shared resources is not atomic however should be exclusive
- Locking is the solution to the race condition
- A lock is associated to one or more shared resource
- Access is granted to a context if it can open the lock



Who Are You Afraid Of?

- ISR can interrupt:
  - Another ISR
  - RTOS
  - Task
- RTOS can preempt a task
- A task can preempt a task
- Different locks needed



Disabling Interrupts

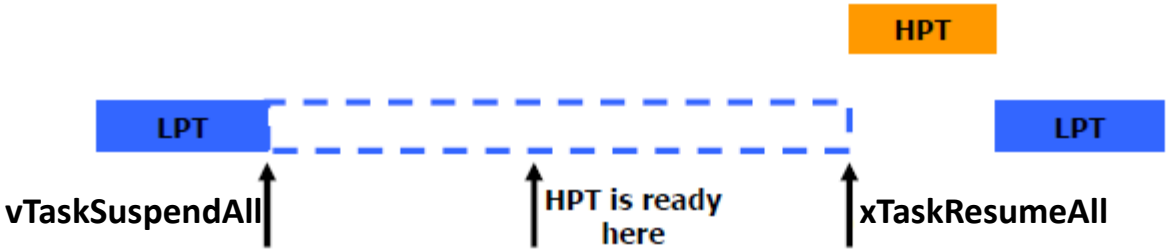
- 2 macros provided taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS()
- Disable/Enable global interrupts
  - If configMAX_SYSCALL_INTERRUPT_PRIORITY not used, all interrupts disabled
  - If configMAX_SYSCALL_INTERRUPT_PRIORITY used, all interrupts with pririties up to configMAX_SYSCALL_INTERRUPT_PRIORITY are disabled
- Not recommended to call FreeRTOS services within a critical section
- They do not support nesting
- taskENTER_CRITICAL() and taskEXIT_CRITICAL() support nesting

Example: Disabling Interrupts

```
void vDemoFunction( void )
{
    taskENTER_CRITICAL();
    /* Perform critical section */
    taskEXIT_CRITICAL();
}

void vTask1( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some functionality here. */
        taskENTER_CRITICAL();
        /* Perform critical section */
        vDemoFunction();
        /* Perform rest of critical section */
        taskEXIT_CRITICAL();
    }
}
```


Disabling Scheduler

- 2 macros provided vTaskSuspendAll() and xTaskResumeAll()
- They can nest
- Not recommended to call FreeRTOS services within a critical section

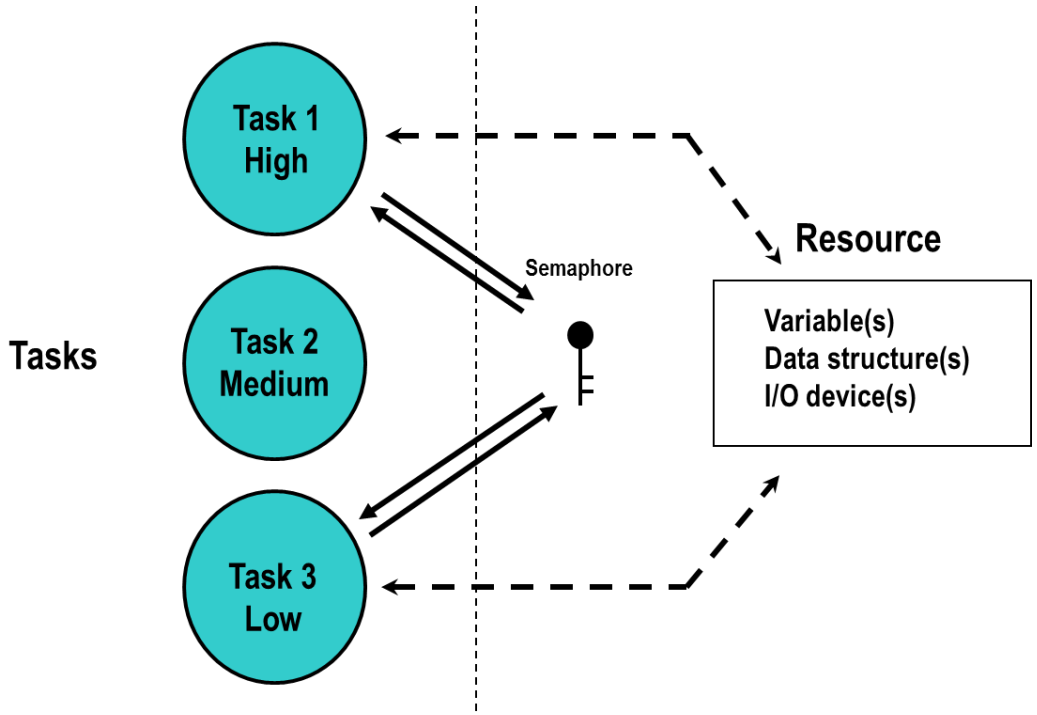


Semaphore = Unsigned Counters

- Can be counting or binary
- Has 2 operations
  - Give = Increment counter and unblock
  - Take = If counter > 0, decrement counter; otherwise block caller
- Semaphore can be used for both access and event synchronization

Semaphore Type	Access Synchronization (Initially > 0)	Even Synchronization (Initially = 0)
Counting	Resource limit	??
Binary	Shared resource protection	??

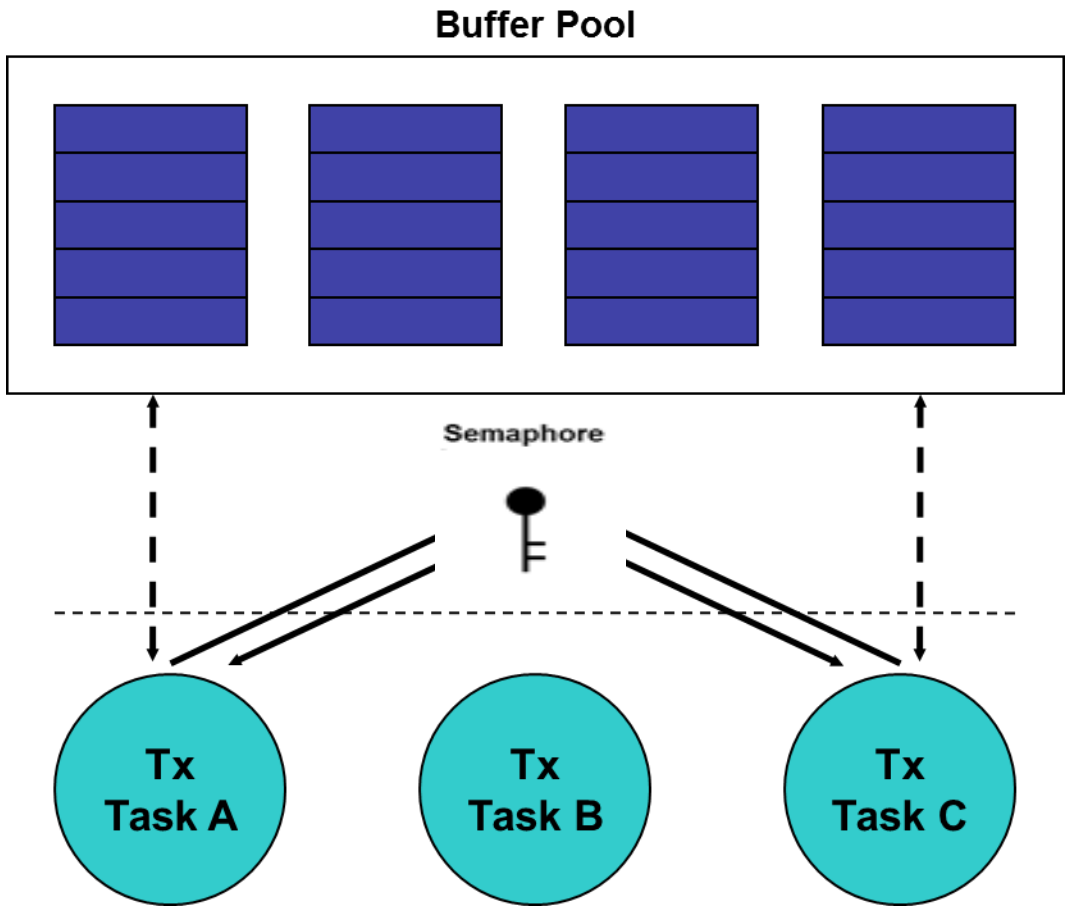
Example: Shared Resource Protection



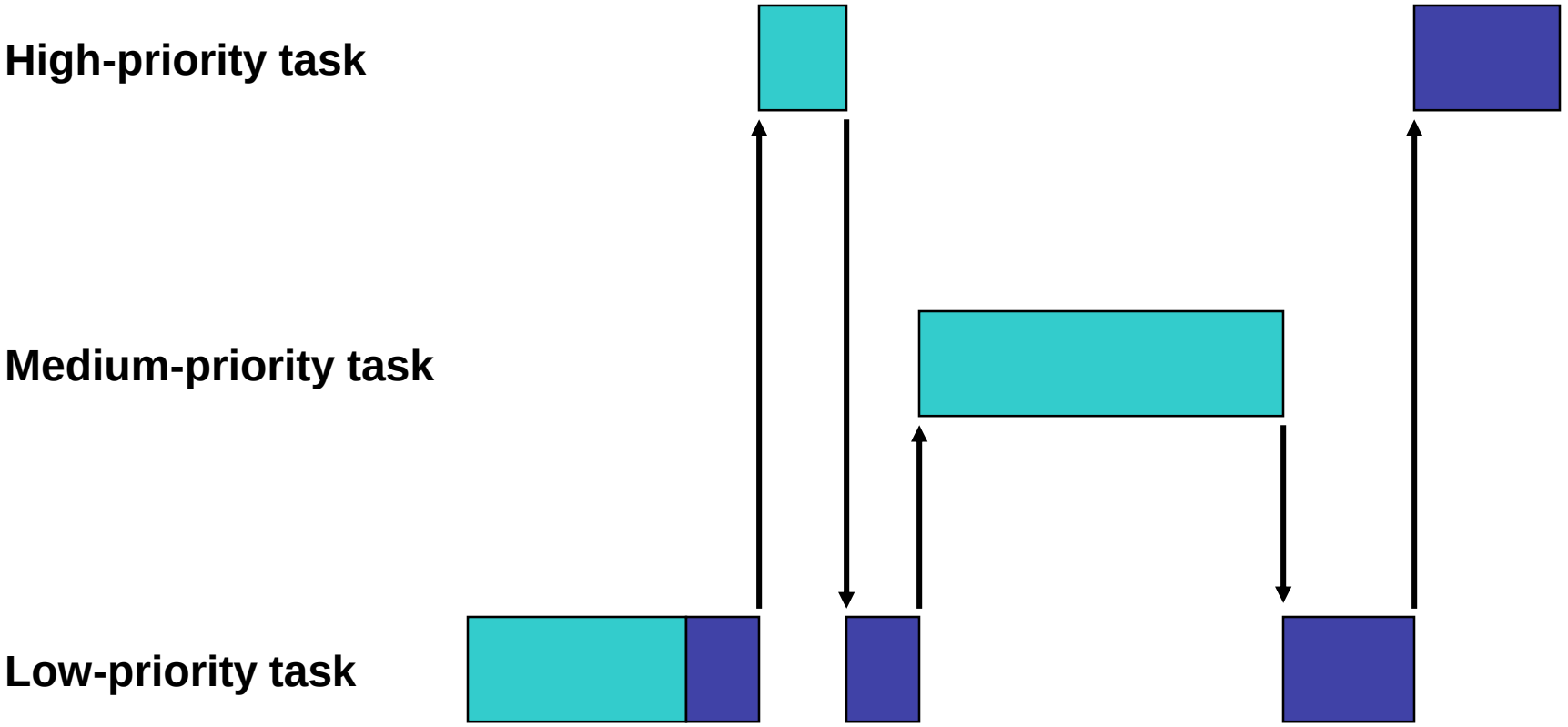
Example: Shared Resource Protection

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask(void * pvParameters)
{

    xSemaphore = xSemaphoreCreateBinary();
    if(xSemaphore == NULL)
    {
        /* Could not create */
    }
    /* To initialize binary semaphore to 1 */
    xSemaphoreGive(xSemaphore);

    for(;;)
    {
        if( xSemaphoreTake(xSemaphore, (TickType_t )0))
        {
            /* Shared resource access */
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                /* Should not fail */
            }
        }
    }
}
```


Example: Resource Limit

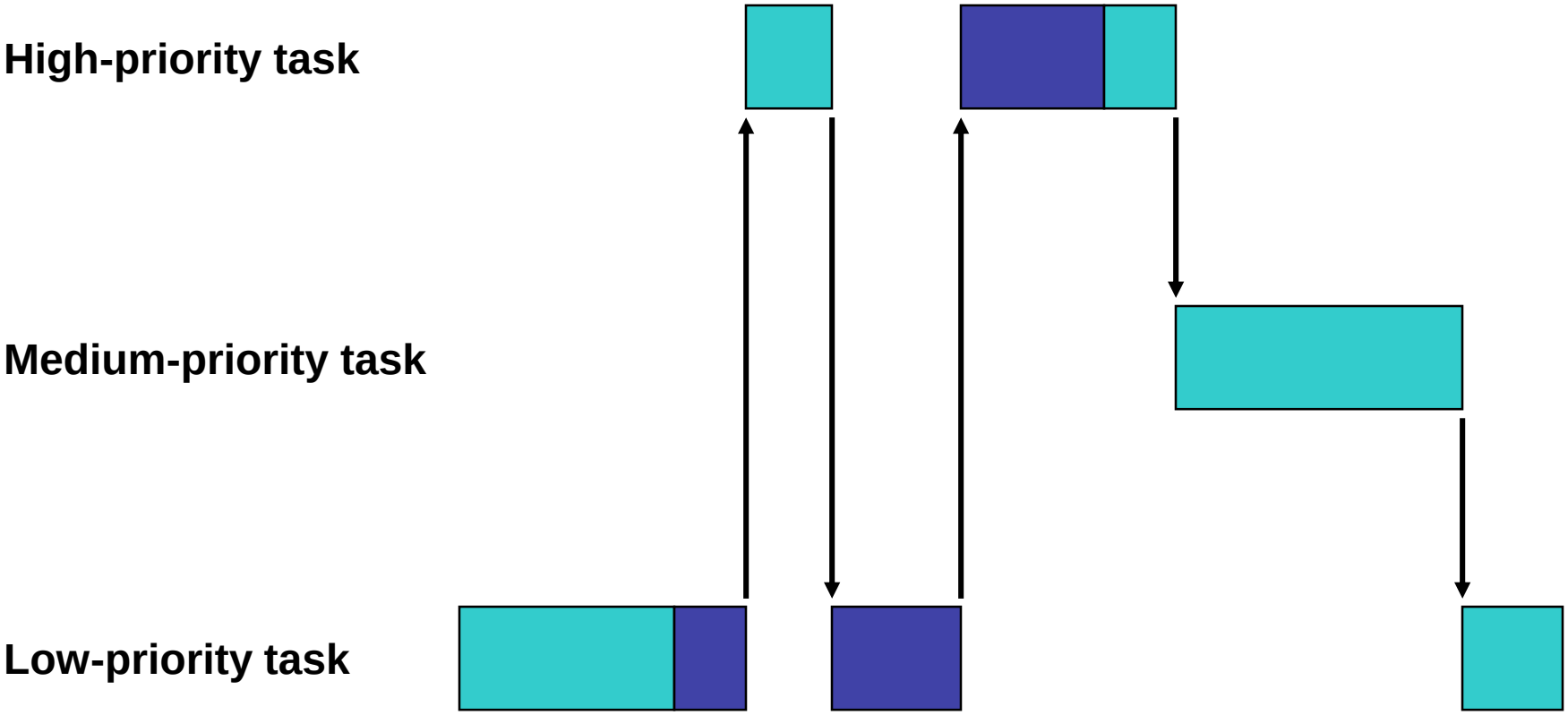


Semaphore Problem – Priority Inversion



Priority Inheritance

- Not all priority inversion is removed, a thorough design might be needed

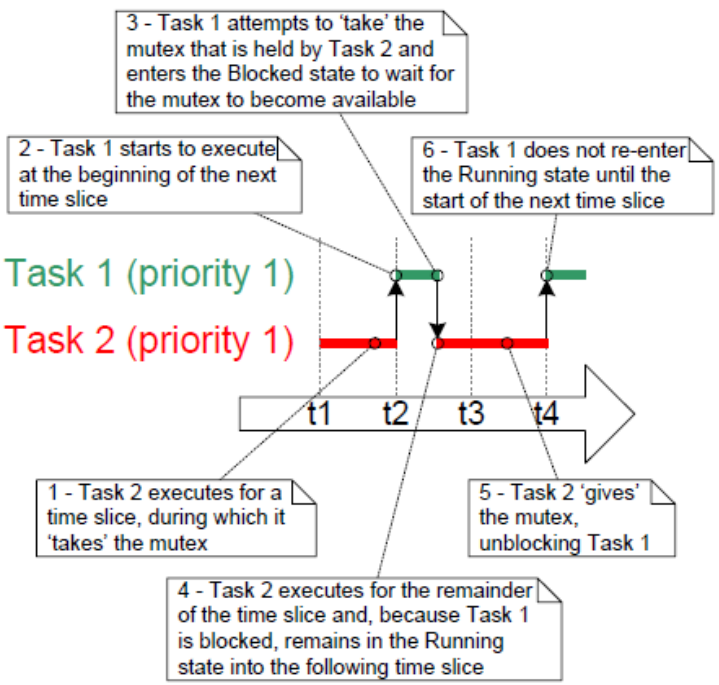


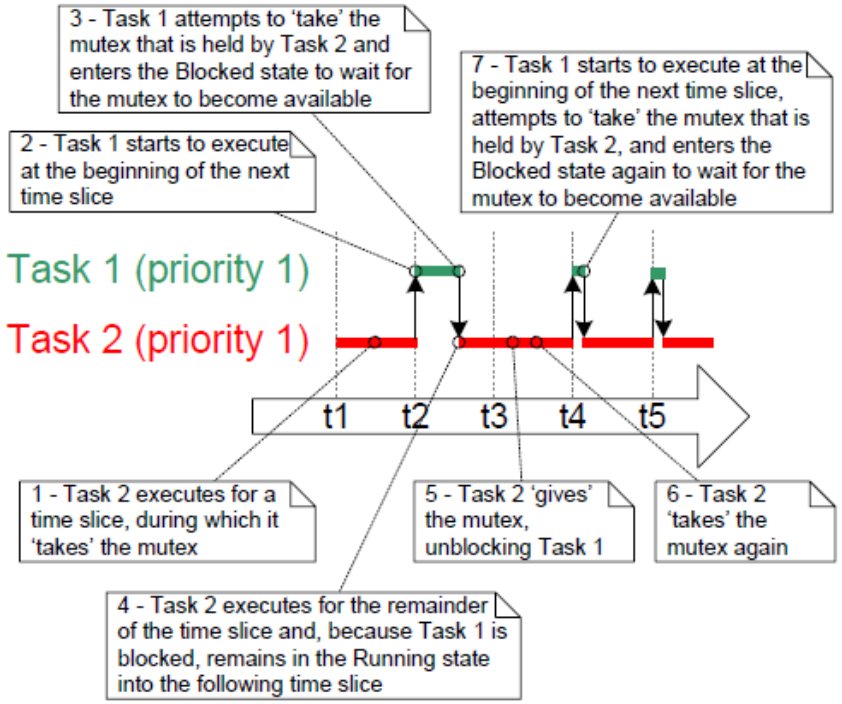
Mutex

- A mutex is yet another mechanism for protecting shared resources
- Mutex is a binary semaphore but w/ built-in protection from priority inversion
- Unlike binary semaphore, a mutex:
  - Has an owner
  - Can not be used from ISR
  - Can be recursive
  - Not used in event synchronization

Mutex and Task Scheduling

- A Mutex give will unblock and give control to higher priority tasks immediately
- Care when mutex unblocks a task of same priority

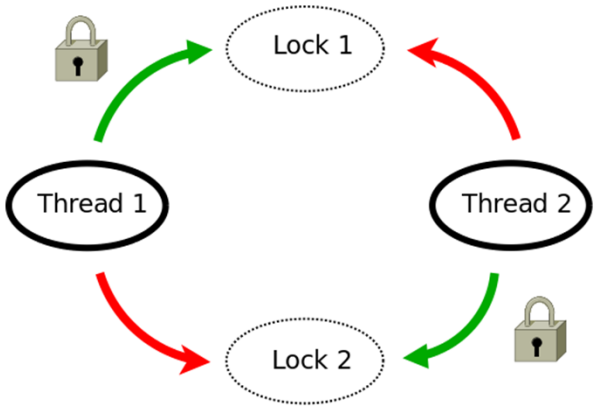




Semaphore/Mutex Problem – Deadlock

- Coffmann conditions
  - Mutual exclusion
  - Hold and wait
  - Circular wait
  - No preemption

- Deadlocks can be
  - Prevented
  - Avoided
  - Recovered



Deadlock Solutions

- Prevention is done by ensuring @ design time, 1 of Coffmann conditions is not achieved
  - Should be primary mechanism
- Avoidance is ensuring @ run-time, no circular dependencies happen
  - Resource manager @ run-time
  - Overhead for real-time embedded systems
- Recovery needs detection and removal mechanisms @ run-time
  - WDT for example
  - Should be used as a last result

Deadlock Prevention Techniques

- Ordered locking of different shared resources (eliminates circular wait)
- Shared resource handler that queue requests to a shared resource (eliminates mutual exclusion)
  - No risk or priority inversion or deadlock
  - Can be used w/ interrupts as well
- Shared resource manager that allow request pre-emption in some cases (eliminates no preemption)
- Allocate and use one shared resource @ a time (eliminate hold and wait and circular wait)
- Allocate all required shared resources simultaneously (eliminate hold and wait and circular wait)
- Allocate on request of shared resource (eliminate hold and wait and circular wait)

Example: Shared Resource Handler (Gate Keeper)

```
static void prvStdioGatekeeperTask(void *pvParameters)
{
    char *pcMessageToPrint;
    /* This is the only task that is allowed to access shared resource */

    for( ;; )
    {
        /* Wait for a message to arrive indefinitely */
        xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX_DELAY);
        /* Access Shared resource on behalf of requester */
    }
}

static void prvPrintTask(void *pvParameters)
{
    /* Initialization */
    for( ;; )
    {
        /* Send message to Gate Keeper, there should be enough space in queue */
        xQueueSendToBack(xPrintQueue, &(pcStringsToPrint[iIndexToString]), 0);
        /* Do something else */
    }
}
```


FreeRTOS Semaphore/Mutex APIs

Creation	Control	Utilities
xSemaphoreCreateBinary xSemaphoreCreateBinaryStatic vSemaphoreCreateBinary xSemaphoreCreateCounting xSemaphoreCreateCountingStatic xSemaphoreCreateMutex xSemaphoreCreateMutexStatic xSemaphoreCreateRecursiveMutex xSemaphoreCreateRecursiveMutexStatic vSemaphoreDelete	xSemaphoreTake xSemaphoreTakeRecursive xSemaphoreGive xSemaphoreGiveRecursive	xSemaphoreGetMutexHolder uxSemaphoreGetCount

Creating a Binary Semaphore, xSemaphoreCreateBinary

```
SemaphoreHandle_t xSemaphoreCreateBinary(void);
vSemaphoreCreateBinary(SemaphoreHandle_t xSemaphore);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- Create an empty binary semaphore (counter = 0)
- Return binary semaphore handle or NULL
- vSemaphoreCreateBinary() is deprecated and kept for compatibility

Creating a Binary Semaphore, xSemaphoreCreateBinaryStatic

```
SemaphoreHandle_t xSemaphoreCreateBinaryStatic(
                  StaticSemaphore_t *pxSemaphoreBuffer);
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pxSemaphoreBuffer will be used to hold binary semaphore data
- Return static binary semaphore handle or NULL

Creating a Counting Semaphore, xSemaphoreCreateCounting

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount,
                                           UBaseType_t uxInitialCount);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- Return counting semaphore handle or NULL

Creating a Counting Semaphore, xSemaphoreCreateCountingStatic

```
SemaphoreHandle_t xSemaphoreCreateCountingStatic(UBaseType_t uxMaxCount,
        UBaseType_t uxInitialCount, StaticSemaphore_t *pxSemaphoreBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pxSemaphoreBuffer will be used to hold counting semaphore data
- Return static counting semaphore handle or NULL

Creating a Mutex, xSemaphoreCreateMutex

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- Return mutex handle or NULL

Creating a Mutex, xSemaphoreCreateMutexStatic

```
SemaphoreHandle_t xSemaphoreCreateMutexStatic(
        StaticSemaphore_t *pxMutexBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pxMutexBuffer will be used to hold mutex data
- Return static mutex handle or NULL

Creating a Recursive Mutex, xSemaphoreCreateRecursiveMutex

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- configUSE_RECURSIVE_MUTEXES must be 1
- Return recursive mutex handle or NULL

Creating a Recursive Mutex, xSemaphoreCreateRecursiveMutexStatic

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic(
                                    StaticSemaphore_t *pxMutexBuffer);
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- configUSE_RECURSIVE_MUTEXES must be 1
- pxMutexBuffer will be used to hold recursive mutex data
- Return static recursive mutex handle or NULL

Deleting a Semaphore, vSemaphoreDelete

```
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);
```

- Delete a binary semaphore, counting semaphore, mutex or recursive mutex
- Do not delete a semaphore that has tasks blocked on it

Taking a Semaphore/Mutex

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,
        TickType_t xTicksToWait);
BaseType_t xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex,
        TickType_t xTicksToWait);
```

- xTicksToWait is blocking timeout in ticks
  - If zero, semaphore/mutex is polled
  - If INCLUDE_vTaskSuspend is 1, using portMAX_DELAY as timeout will cause indefinite block
- Return pdTRUE if taken, pdFALSE if timeout

Giving a Semaphore/Mutex

```
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
BaseType_t xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex);
```

- configUSE_RECURSIVE_MUTEXES must be 1 for xSemaphoreGiveRecursive()
- Return pdTRUE if successful, pdFALSE if an error

Other Semaphore/Mutex Getters

```
TaskHandle_t xSemaphoreGetMutexHolder(SemaphoreHandle_t xMutex);
```

- INCLUDE_xSemaphoreGetMutexHolder must be 1
- configUSE_MUTEXES must be 1
- Mutex holder might change between the call and testing the return value
- Return task handle of owns mutex or NULL if error

```
UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore);
```


Exercise: Inter-task Access Synchronization

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- **Inter-task Event Synchronization**
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Task Interaction

- RTOS tasks are not necessarily self-contained
- In order for the application's objectives to be met, tasks may need to interact with each other (and possibly with ISRs).
- A typical RTOS provides services that facilitate such interaction

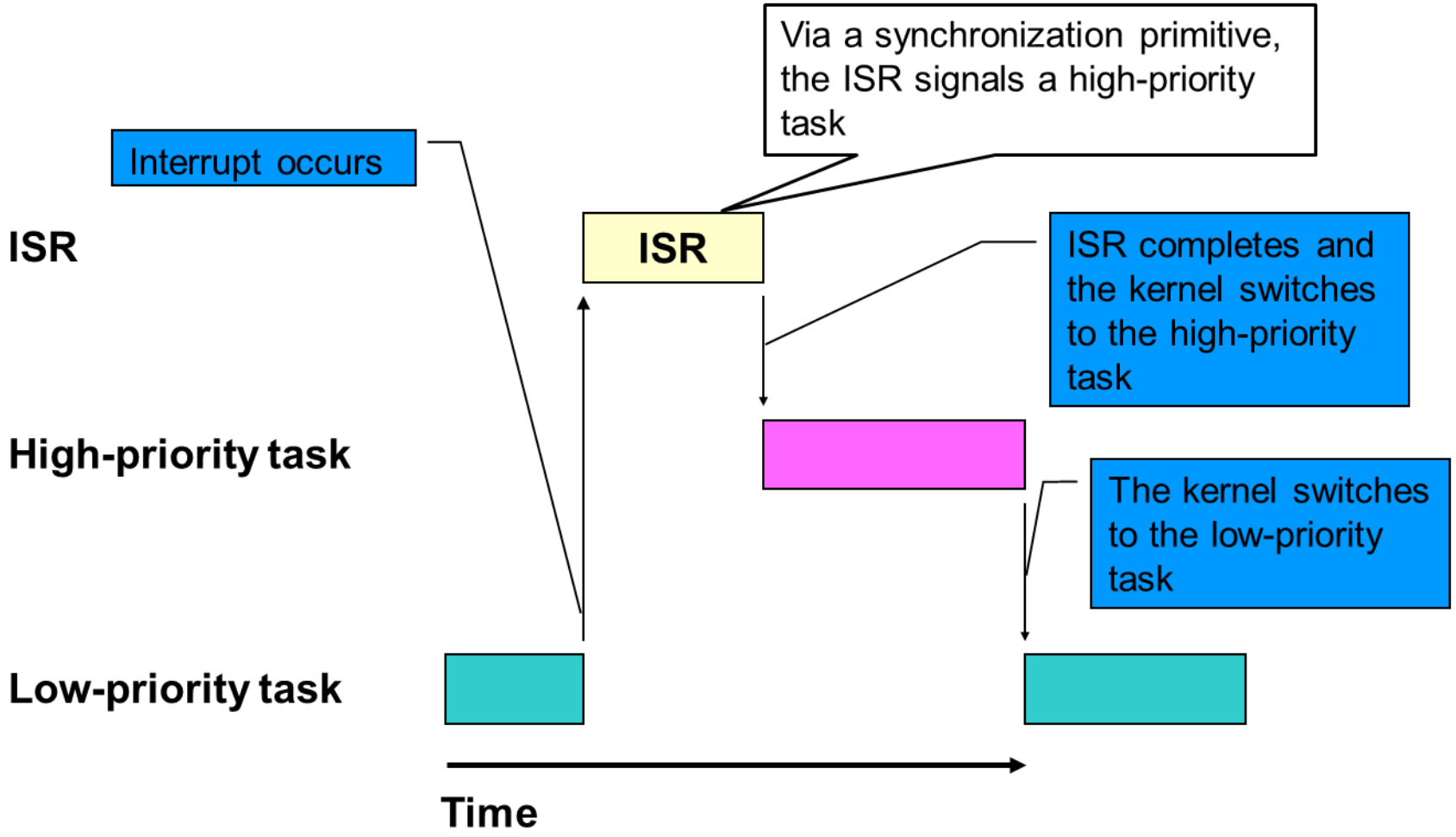
Lengthy ISRs

- On many architectures, a long ISR can significantly increase interrupt latency
- Excessively large stacks may be needed in order to support lengthy ISRs
- Debugging interrupt handlers can be difficult
- Many kernel functions cannot be invoked by ISRs

Synchronizing a Task to an ISR

- Most applications must manage a collection of peripheral devices
- The interrupt service routines (ISRs) associated with a system's peripheral devices should be kept brief
- In applications that incorporate a real-time kernel, ISRs can use synchronization primitives to signal tasks
- Using a semaphore, a task can synchronize to another task or to an ISR

Synchronizing a Task to an ISR cont'd

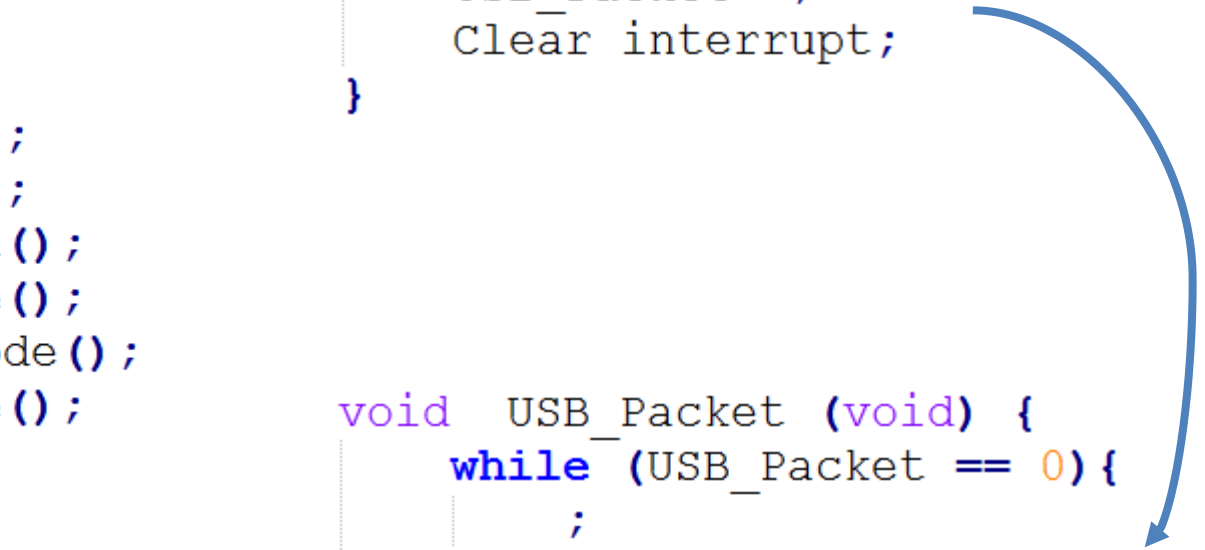


Signaling in Foreground/Background Systems

```
while (1) {
    ADC_Read();
    SPI_Read();
    USB_Packet();
    LCD_Update();
    Audio_Decode();
    File_Write();
}
```

```
void  USB_ISR (void) {
    USB_Packet++;
    Clear interrupt;
}
```

```
void  USB_Packet (void) {
    while (USB_Packet == 0){
        ;
    }
    Process packet;
}
```

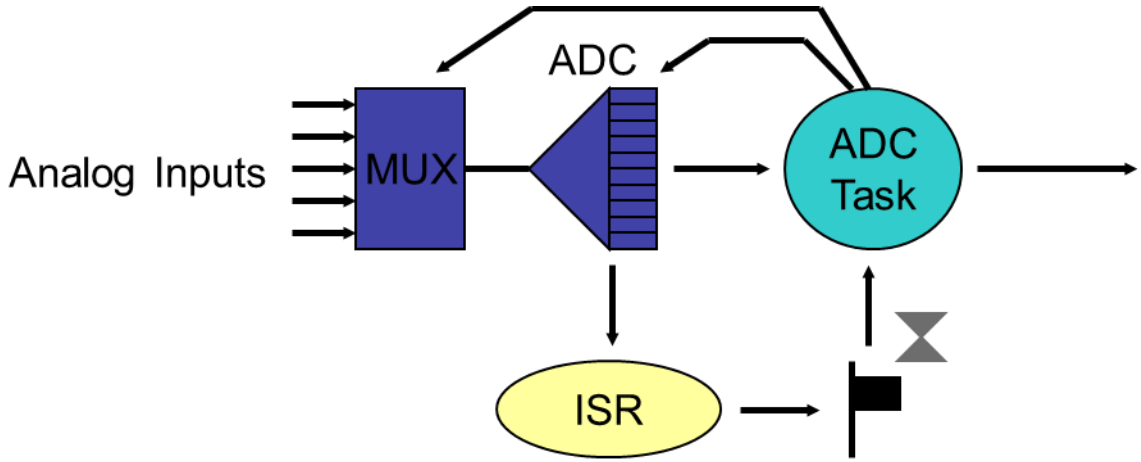


Semaphore = Unsigned Counters Revisited

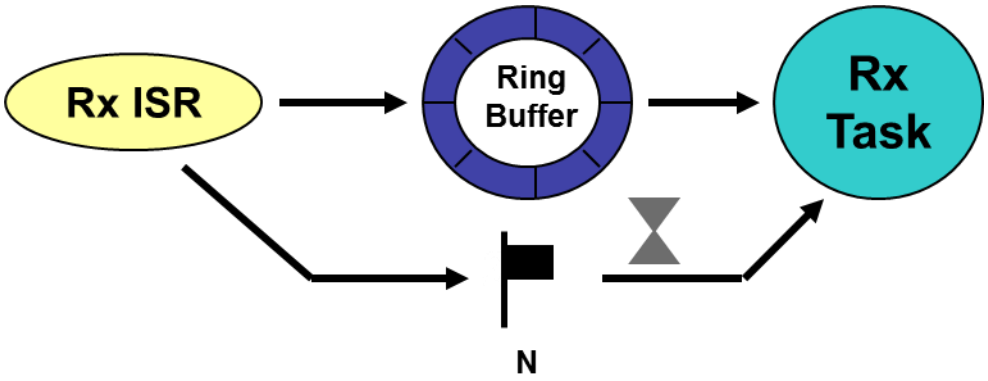
- Semaphore can be used for both access and event synchronization

Semaphore Type	Access Synchronization (Initially > 0)	Even Synchronization (Initially = 0)
Counting	Resource limit	Producer - Consumer
Binary	Shared resource protection	Wait - Signal

Example: Wait – Signal

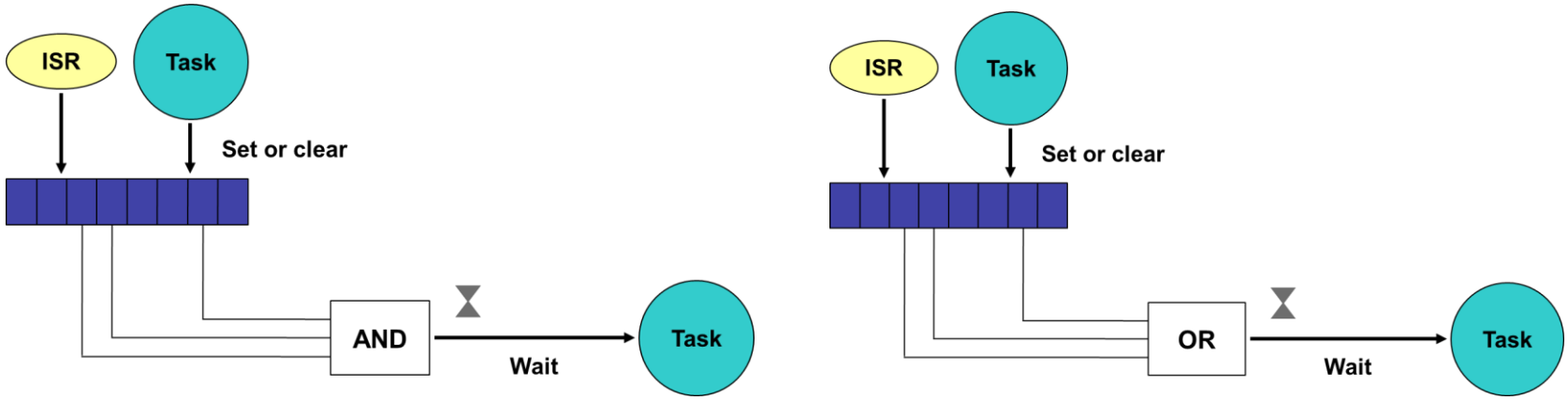


Example: Producer – Consumer

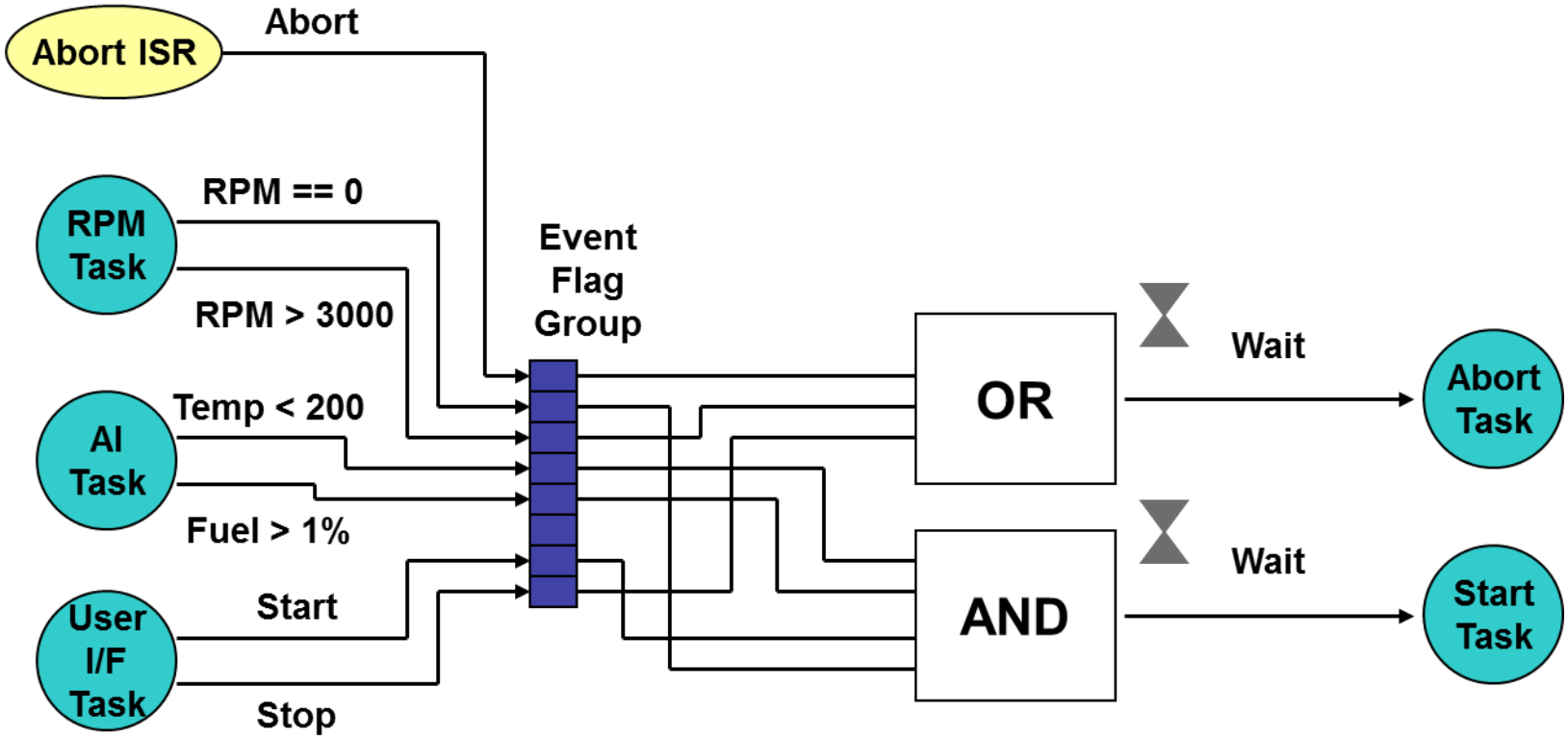


Event Flags and Event Groups

- Using event flags, a task can easily wait for multiple events to take place
- A single 8-, or 24-bit variable, contained in a structure known as an event flag group, represents a collection of events
  - 8 if configUSE_16_BIT_TICKS is 1, or 24 if set to 0



Example: Event Flags



FreeRTOS Events APIs

Creation	Control	Utilities
xEventGroupCreate xEventGroupCreateStatic vEventGroupDelete	xEventGroupWaitBits xEventGroupSetBits xEventGroupClearBits xEventGroupSync	xEventGroupGetBits

Creating/Deleting an Event Group

```
EventGroupHandle_t xEventGroupCreate(void);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- Return event group handle or NULL

```
EventGroupHandle_t xEventGroupCreateStatic(
        StaticEventGroup_t *pxEventGroupBuffer);
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pxEventGroupBuffer will be used to hold event group data

```
void vEventGroupDelete(EventGroupHandle_t xEventGroup);
```

- Tasks blocked on event will be unblocked and report an event group value of 0

Waiting for Event Flags, xEventGroupWaitBits

```
EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup,
        const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit,
        const BaseType_t xWaitForAllBits, TickType_t xTicksToWait );
```

- uxBitsToWaitFor is mask of event flags to wait for and must not be 0
- xClearOnExit if pdTRUE then the uxBitsToWaitFor masked bits are cleared before return if return for any reason other than a timeout.
- xClearOnExit if pdFALSE then the bits set in the event group are not altered before return
- xWaitForAllBits if pdTRUE it is AND-wait, if false it is OR-wait
- Return value can be used to test successful wait or timeout if xClearOnExit is pdFALSE
- Return value may change between the return and the test

Managing Event Flags

```
EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup,
                               const EventBits_t uxBitsToSet);
```

- Setting bits will unblock any tasks were blocked
- Return value may change between the return and the test

```
EventBits_t xEventGroupClearBits(EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToClear);
```

- Return event group value @ call

```
EventBits_t xEventGroupGetBits(EventGroupHandle_t xEventGroup);
```

- Return event group value @ call

Synching to an Event Group, xEventGroupSync

```
EventBits_t xEventGroupSync(EventGroupHandle_t xEventGroup,
                            const EventBits_t uxBitsToSet, const EventBits_t uxBitsToWaitFor,
                            TickType_t xTicksToWait);
```

- Atomically set event flags in an event group, then wait for a combination of event flags to be set within the same event group
- Used for rendezvous synchronization
- Return event group value @ time bits being waited for became set or timeout
- Return value may change between the return and the test

Example: Rendezvous

```
/* Bits used by the three tasks. */
#define TASK_0_BIT        ( 1 << 0 )
#define TASK_1_BIT        ( 1 << 1 )
#define TASK_2_BIT        ( 1 << 2 )
#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

/* Use an event group to synchronize three tasks */
EventGroupHandle_t xEventBits;

void vTaski( void *pvParameters )
{
    EventBits_t uxReturn;
    TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    for( ;; )
    {
        ...
        uxReturn = xEventGroupSync(xEventBits, TASK_i_BIT,
                    ALL_SYNC_BITS, xTicksToWait);

        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
            /* All tasks rendezvous */
        }
    }
}
```


Exercise: Inter-task Event Synchronization

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- **Inter-task Communication**
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Inter-Task Communication Services

- Tasks in RTOS-based application can send and receive messages using services that FreeRTOS provides

- Application developers determine the contents of these messages

```
void  App_TaskFIR (void *p_arg){
    while (1) {
        Read next sample;
        Calculate filter output;
        Send output value to App_TaskLog();
    }
}
```

```
void  App_TaskLog (void *p_arg){
    while (1) {
        Receive output from App_TaskFIR();
        Write filter output to file;
    }
}
```

- Message passing services have much in common with event/access synchronization

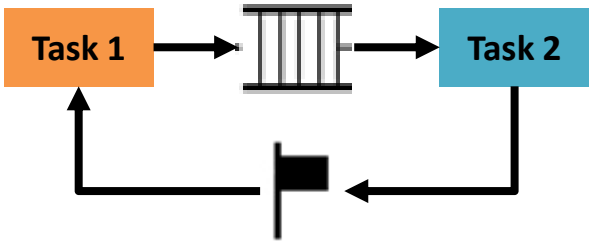
FreeRTOS Queues

- Task-safe FIFO buffers
- Messages placed by copying not reference
- Fixed size messages defined during creation
- Messages can be references, variable size messages or different message types (application programer responsibility)
- Queues can block sender (if queue is full) or receiver if (queue is empty) but to a timeout
- Implementation is suitable for MPU/MMU powered environments

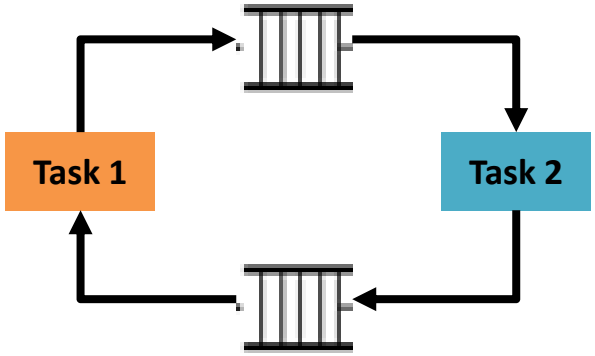
Queues Use Cases

1-Way Non-Interlocked

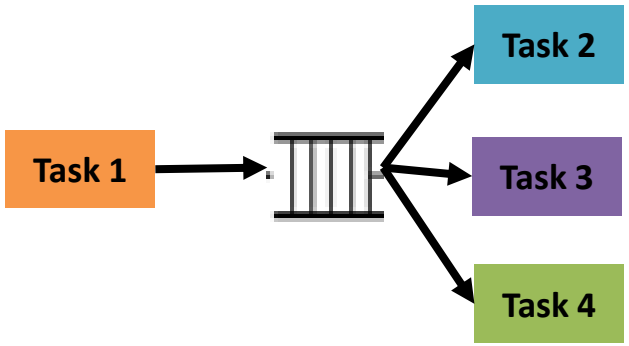
1-Way Interlocked



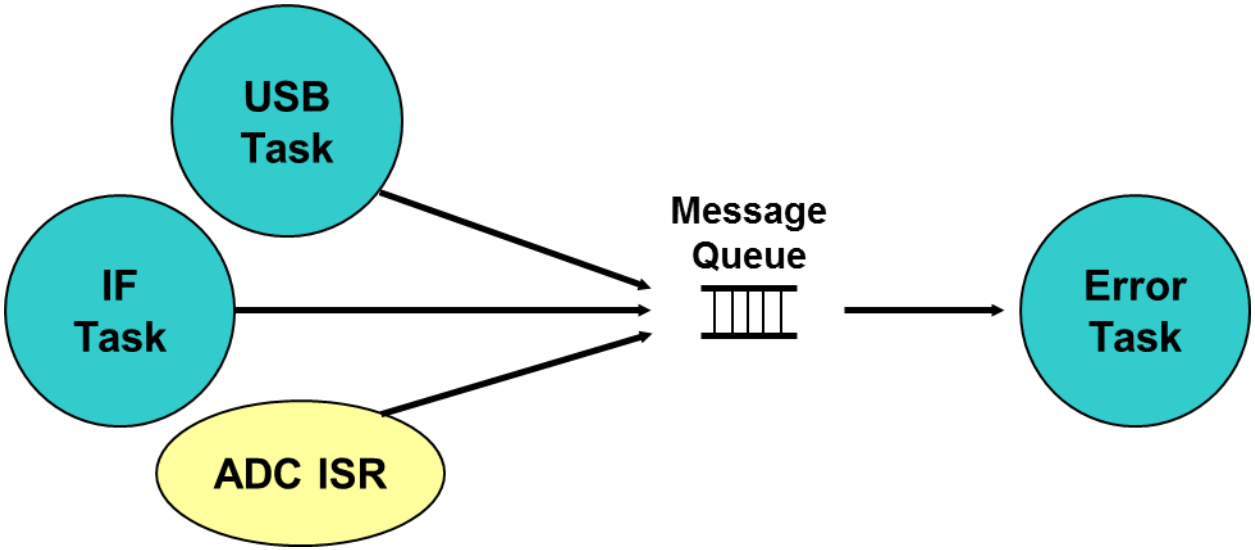
2-Way Interlocked



Broadcast



Many to 1 Communication



Example: Many to 1 Communication

```
/* Define type to identify source of data */
typedef enum
{
    eSender1,
    eSender2
} DataSource_t;

/* Define the message structure */
typedef struct
{
    uint8_t ucValue;
    DataSource_t eDataSource;
} Data_t;

/* Declare 2 messages */
static const Data_t xStructsToSend[ 2 ] =
{
    {100, eSender1}, /* Used by Sender1. */
    {200, eSender2} /* Used by Sender2. */
};
```


Example: Many to 1 Communication cont'd

```
static void vSenderTask( void *pvParameters )
{
    BaseType_t xStatus;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    for( ;; )
    {
        xStatus = xQueueSendToBack(xQueue, pvParameters, xTicksToWait );
        if(xStatus != pdPASS )
        {
            /* Handle error */
        }
    }
}
```


Example: Many to 1 Communication cont'd

```
static void vReceiverTask(void *pvParameters)
{

    Data_t xReceivedStructure;
    BaseType_t xStatus;

    for(;;)
    {
        xStatus = xQueueReceive(xQueue, &xReceivedStructure, 0);
        if(xStatus == pdPASS)
        {
            /* Data was successfully received , print it */
            if(xReceivedStructure.eDataSource == eSender1)
            {
                /* Source one */
            }
            else
            {
                /* source 2 */
            }
        }
    }
}
```


Example: Many to 1 Communication cont'd

```
int main( void )
{
    xQueue = xQueueCreate(3, sizeof( Data_t));
    if(xQueue != NULL )
    {
        xTaskCreate(vSenderTask, "Sender1", 1000, &(xStructsToSend[ 0 ]), 2, NULL);
        xTaskCreate(vSenderTask, "Sender2", 1000, &(xStructsToSend[ 1 ]), 2, NULL);

        xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 1, NULL);
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    for( ;; );
}
```


FreeRTOS Queues APIs

Creation	Control	Utilities
xQueueCreate xQueueCreateStatic vQueueDelete	xQueueSend xQueueSendToBack xQueueSendToFront xQueueReceive xQueuePeek xQueueOverwrite	xQueueReset uxQueueMessagesWaiting uxQueueSpacesAvailable vQueueAddToRegistry pcQueueGetName vQueueUnregisterQueue

Creating a Queue, xQueueCreate

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- uxQueueLength is queue size in message
- uxItemSize is single message size in bytes
- Return queue handle or NULL

```
QueueHandle_t xQueueCreateStatic(UBaseType_t uxQueueLength,
        UBaseType_t uxItemSize, uint8_t *pucQueueStorageBuffer,
        StaticQueue_t *pxQueueBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pucQueueStorageBuffer is queue storage and should be ≥ uxQueueLength * uxItemSize
- pxQueueBuffer will be used to hold event group data

Deleting a Queue, vQueueDelete

```
void vQueueDelete(TaskHandle_t pxQueueToDelete);
```

- Do not delete a queue that has tasks blocked on it
- Can be used to delete a semaphore also

Sending to a Queue

```
BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue,
        TickType_t xTicksToWait);
BaseType_t xQueueSendToFront(QueueHandle_t xQueue, const void * pvItemToQueue,
        TickType_t xTicksToWait );
BaseType_t xQueueSendToBack(QueueHandle_t xQueue, const void * pvItemToQueue,
        TickType_t xTicksToWait );
```

- xQueueSend() == xQueueSendToBack() ➔ FIFO
- xQueueSendToFront() ➔ LIFO
- Return pdTRUE if successful , errQUEUE_FULL otherwise

Receiving from a Queue

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer,
        TickType_t xTicksToWait );
```

- Destructive read
- Return pdTRUE if successful , errQUEUE_EMPTY otherwise

```
BaseType_t xQueuePeek(QueueHandle_t xQueue, void *pvBuffer,
        TickType_t xTicksToWait );
```

- Non-destructive read

Overwriting/Resetting a Queue

```
BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void *pvItemToQueue);
```

- Like xQueueSendToBack() but write to queue even if full overwriting data
- Intended for use w/ queues that have a length of one
- Return pdPASS only

```
BaseType_t xQueueReset(QueueHandle_t xQueue);
```

- Resets a queue to its original empty state
- Return pdPASS only

Other Queue APIs

```
UBaseType_t uxQueueMessagesWaiting(const QueueHandle_t xQueue);
UBaseType_t uxQueueSpacesAvailable(const QueueHandle_t xQueue);


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

- Queue registery assigns human readable names for queues and semaphores
- Used w/ kernel-aware debuggers
- configQUEUE_REGISTRY_SIZE define max number of registry entries
- Deleting a queue will automatically remove it from the registry.

```
const char *pcQueueGetName(QueueHandle_t xQueue);
```

- Return queue name if found, NULL otherwise

FreeRTOS Stream/Message Buffers

- Intertask communication object optimized for single write single reader scenario (Task 2 Task, ISR 2 Task or CPU Core to CPU core)
- Not safe to have multiple different writers/readers
  - If needed protect reading/writing operation as a critical section
- Data is passed by copy
- Stream buffers pass a continuous stream of bytes
- Stream buffers are built on top of direct to task notification (they can change task notification values if their user blocks)
- Message buffers = Stream buffers + variable length discrete messages

More about FreeRTOS Stream/Mesage Buffers

- Stream buffer reader blocks if it is empty and a timeout is specified
- Trigger level defines # of bytes must be available before a reader unblocks
  - 1 ≤ Trigger level ≤ Max stream buffer size
- If a blocked reader times out before trigger level reached, reader will read available bytes
- Stream buffer writer block if full and a timeout specified
- configMESSAGE_BUFFER_LENGTH_TYPE define the type used to store the message length (if not defined, defaults to size_t)
- Message = Data + Message length

Core to Core Communication

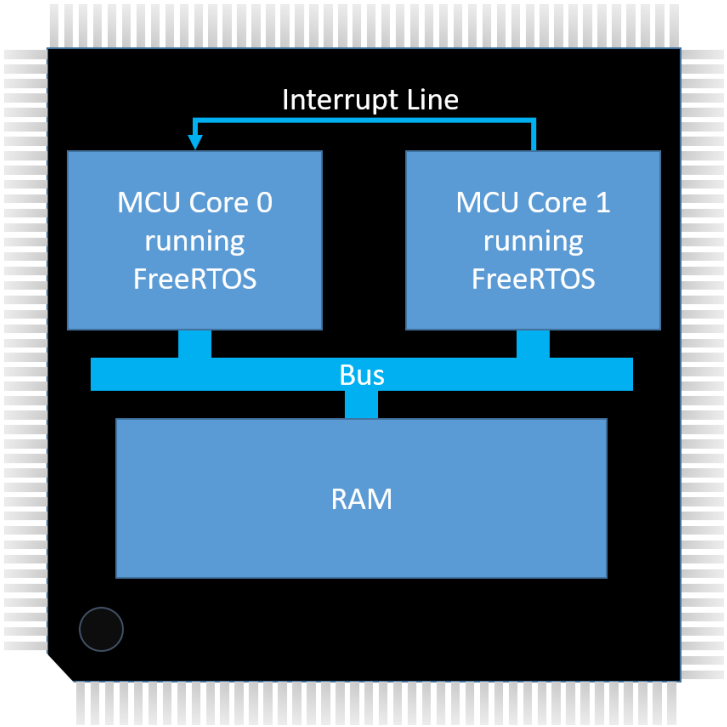
- sbSEND_COMPLETED()/sbRECEIVE_COMPLETED() are FreeRTOS internal macros and located in FreeRTOSConfig.h
- They take the stream buffer handle
- sbSEND_COMPLETED() checks if a writer is blocked and unblocks it
- sbRECEIVE_COMPLETED() is receive equivalent of sbSEND_COMPLETED()
- Change their implementation to pass data between cores
  - Core 1 can generate an interrupt to core 2 to unblock a task on core 2

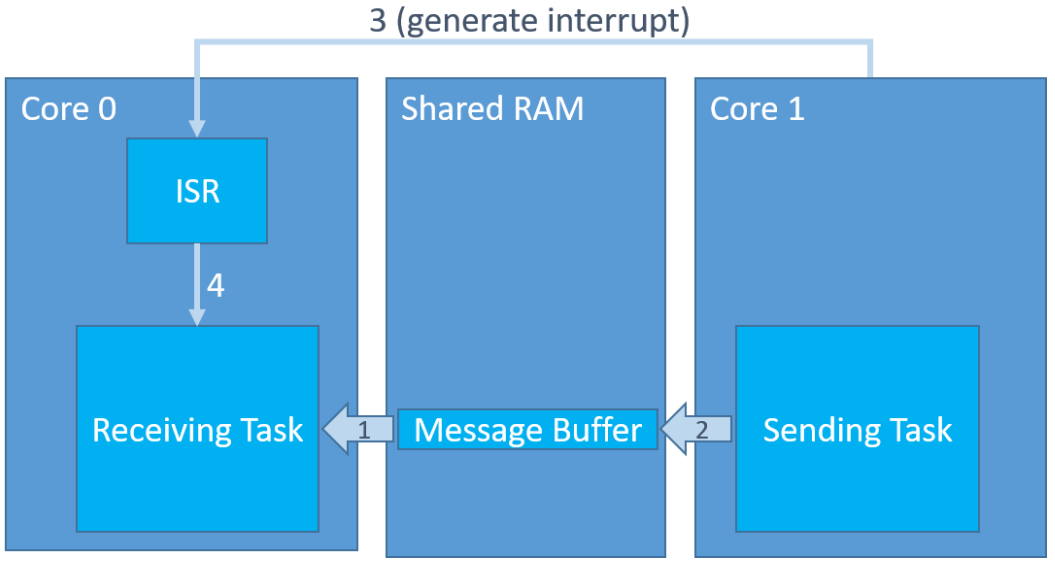
Core to Core Communication cont'd

```
xMessageBufferSend()
{
    /* If a time out is specified and there isn't enough
    space in the message buffer to send the data, then
    enter the blocked state to wait for more space. */
    if( time out != 0 )
    {
        while( there is insufficient space in the buffer &&
               not timed out waiting )
        {
            Enter the blocked state to wait for space in the buffer
        }
    }

    if( there is enough space in the buffer )
    {
        write data to buffer
        sbSEND_COMPLETED()
    }
}
```

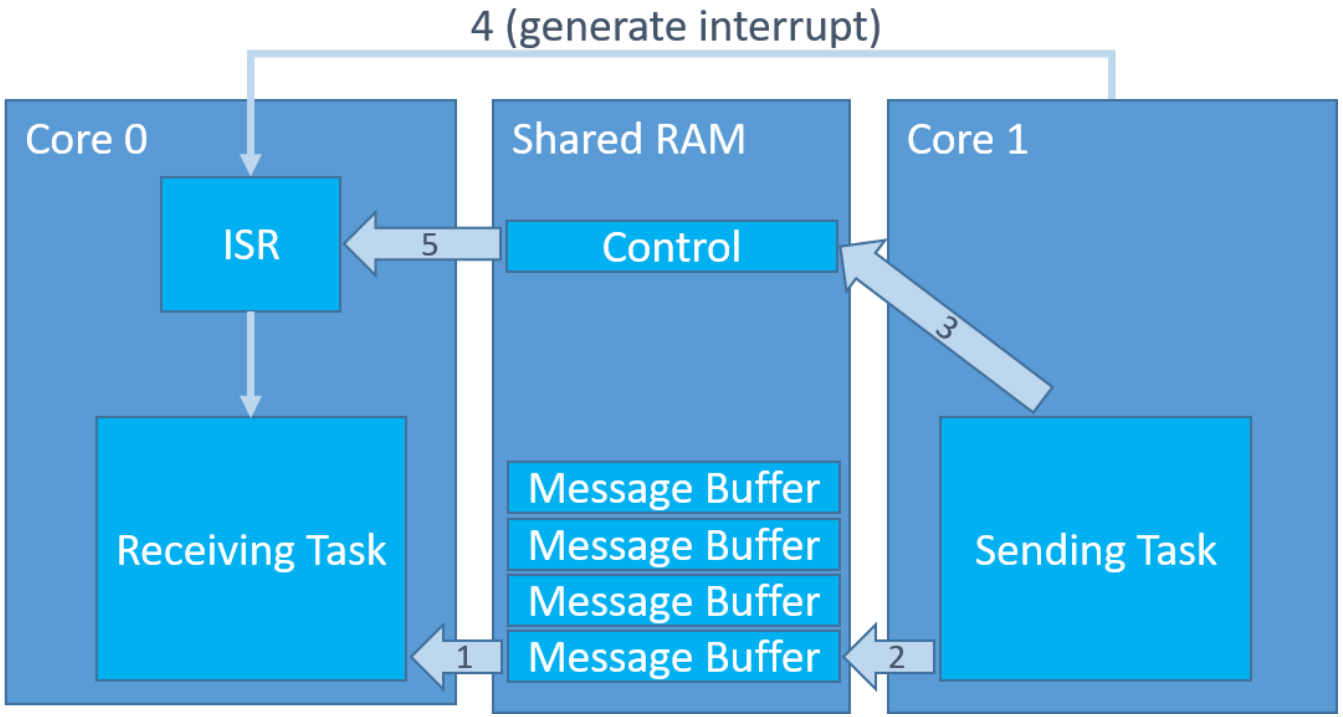
```
Receive()

me out is specified and the buffer doesn't
y data that can be read, then enter the
ate to wait for the buffer to contain data. */
ut != 0 )

 there is no data in the buffer &&
 not timed out waiting )
    {
        Enter the blocked state to wait for data
    }
}

if( there is data in the buffer )
{
    read data from buffer
    sbRECEIVE_COMPLETED()
}
}
```


Example: Core to Core Communication using a single Message Buffer





Example: Core to Core Communication using Multiple Message Buffers cont'd



Example: Core to Core Communication using Multiple Message Buffers cont'd

```
/* Added to FreeRTOSConfig.h to override the default implementation. */
#define sbSEND_COMPLETED( pxStreamBuffer ) vGenerateCoreToCoreInterrupt( pxStreamBuffer )

/* Implemented in a C file. */
void vGenerateCoreToCoreInterrupt( MessageBufferHandle_t xUpdatedBuffer )
{
size_t BytesWritten.

    /* Called by the implementation of sbSEND_COMPLETED() in FreeRTOSConfig.h.
    If this function was called because data was written to any message buffer
    other than the control message buffer then write the handle of the message
    buffer that contains data to the control message buffer, then raise an
    interrupt in the other core.  If this function was called because data was
    written to the control message buffer then do nothing. */
    if( xUpdatedBuffer != xControlMessageBuffer )
    {
        BytesWritten = xMessageBufferSend(  xControlMessageBuffer,
                                            &xUpdatedBuffer,
                                            sizeof( xUpdatedBuffer ),
                                            0 );

        /* If the bytes could not be written then the control message buffer
        is too small! */
        configASSERT( BytesWritten == sizeof( xUpdatedBuffer );

        /* Generate interrupt in the other core (pseudocode). */
        GenerateInterrupt();
    }
}
```


Example: Core to Core Communication using Multiple Message Buffers cont'd

```
void InterruptServiceRoutine( void )
{
MessageBufferHandle_t xUpdatedMessageBuffer;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Receive the handle of the message buffer that contains data from the
    control message buffer.  Ensure to drain the buffer before returning. */
    while( xMessageBufferReceiveFromISR( xControlMessageBuffer,
                                         &xUpdatedMessageBuffer,
                                         sizeof( xUpdatedMessageBuffer ),
                                         &xHigherPriorityTaskWoken )
                                           == sizeof( xUpdatedMessageBuffer ) )
    {
        /* Call the API function that sends a notification to any task that is
        blocked on the xUpdatedMessageBuffer message buffer waiting for data to
        arrive. */
        xMessageBufferSendCompletedFromISR( xUpdatedMessageBuffer,
                                            &xHigherPriorityTaskWoken );
    }

    /* Normal FreeRTOS "yield from interrupt" semantics, where
    xHigherPriorityTaskWoken is initialised to pdFALSE and will then get set to
    pdTRUE if the interrupt unblocks a task that has a priority above that of
    the currently executing task. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```


FreeRTOS Stream Buffers APIs

Creation	Control	Utilities
xStreamBufferCreate xStreamBufferCreateStatic vStreamBufferDelete	xStreamBufferSend xStreamBufferReceive	xStreamBufferBytesAvailable xStreamBufferSpacesAvailable xStreamBufferSetTriggerLevel xStreamBufferReset xStreamBufferIsEmpty xStreamBufferIsFull

Creating/Deleting a Stream Buffer

```
StreamBufferHandle_t xStreamBufferCreate(size_t xBufferSizeBytes,
        size_t xTriggerLevelBytes);
```

- configSUPPORT_DYNAMIC_ALLOCATION must 1 or undefined
- Return stream buffer handle or NULL

```
StreamBufferHandle_t xStreamBufferCreateStatic(size_t xBufferSizeBytes,
        size_t xTriggerLevelBytes, uint8_t *pucStreamBufferStorageArea,
        StaticStreamBuffer_t *pxStaticStreamBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pucStreamBufferStorageArea is stream buffer storage and should be ≥ xBufferSizeBytes + 1
- pxStaticStreamBuffer will be used to hold event group data

```
void vStreamBufferDelete(StreamBufferHandle_t xStreamBuffer);
```


Sending to/Receiving from a Stream Buffer

```
size_t xStreamBufferSend(StreamBufferHandle_t xStreamBuffer,
        const void *pvTxData, size_t xDataLengthBytes,
        TickType_t xTicksToWait );
```

- Return # of bytes written
- If a writer times out before it can write xDataLengthBytes, it will write as many bytes as possible

```
size_t xStreamBufferReceive(StreamBufferHandle_t xStreamBuffer,
        void *pvRxData, size_t xBufferLengthBytes,
        TickType_t xTicksToWait );
```

- Return # of bytes read

Other Stream Buffer APIs

```
size_t xStreamBufferBytesAvailable(StreamBufferHandle_t xStreamBuffer);
```

- Return # of bytes can be read

```
size_t xStreamBufferSpacesAvailable(StreamBufferHandle_t xStreamBuffer);
```

- Return # of bytes can be written

```
BaseType_t xStreamBufferIsFull(StreamBufferHandle_t xStreamBuffer);
```

- Return pdTRUE if full, pdFALSE otherwise

```
BaseType_t xStreamBufferIsEmpty(StreamBufferHandle_t xStreamBuffer);
```

- Return pdTRUE if empty, pdFALSE otherwise

```
BaseType_t xStreamBufferReset(StreamBufferHandle_t xStreamBuffer);
```

- Reset stream buffer if not tasls blocked
- Return pdTRUE if successful, pdFALSE otherwise

FreeRTOS Message Buffers APIs

Creation	Control	Utilities
xMessageBufferCreate xMessageBufferCreateStatic vMessageBufferDelete	xMessageBufferSend xSMessageBufferReceive	xMessageBufferSpacesAvailable xMessageBufferReset xMessageBufferIsEmpty xMessageBufferIsFull

Creating/Deleting a Message Buffer

```
MessageBufferHandle_t xMessageBufferCreate(size_t xBufferSizeBytes);
```

- configSUPPORT_DYNAMIC_ALLOCATION must 1 or undefined
- Return message buffer handle or NULL

```
MessageBufferHandle_t xMessageBufferCreateStatic(size_t xBufferSizeBytes,
    uint8_t *pucMessageBufferStorageArea,
    StaticMessageBuffer_t *pxStaticMessageBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pucMessageBufferStorageArea is message buffer storage and should be ≥ xBufferSizeBytes + 1
- pxStaticMessageBuffer will be used to hold event group data

```
void vMessageBufferDelete(MessageBufferHandle_t xMessageBuffer);
```


Sending to/Receiving from a Message Buffer

```
size_t xMessageBufferSend(MessageBufferHandle_t xMessageBuffer,
    const void *pvTxData, size t xDataLengthBytes, TickType t xTicksToWait);
```

- Writing 10 bytes consumes, 10 bytes + bytes need to store message length
- Return # of bytes written
- If a writer times out before it can write xDataLengthBytes, it return 0

```
size_t xMessageBufferReceive(MessageBufferHandle_t xMessageBuffer,
    void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait );
```

- If xBufferLengthBytes < message length, message is unread
- Return # of bytes read, 0 otherwise

Other Message Buffer APIs

```
size_t xMessageBufferSpacesAvailable(MessageBufferHandle_t xMessageBuffer);
```

- Return # of bytes can be written

```
BaseType_t xMessageBufferIsFull(MessageBufferHandle_t xMessageBuffer);
```

- Return pdTRUE if full, pdFALSE otherwise

```
BaseType_t xMessageBufferIsEmpty(MessageBufferHandle_t xMessageBuffer);
```

- Return pdTRUE if empty, pdFALSE otherwise

```
BaseType_t xMessageBufferReset(MessageBufferHandle_t xMessageBuffer);
```

- Reset stream buffer if not tasls blocked
- Return pdTRUE if successful, pdFALSE otherwise

Exercise: Inter-task Communication

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- **Direct to Task Notification**
- SW Timers
- Memory Management
- Interrupt Management
- Miscellaneous Topics

Direct to Task Notification

- No intermediate communication objects (45% faster and uses less RAM compared to binary semaphore)
- Each task has direct 32-bit notification value initially 0
- Task notification value can be updated in the following ways
  - Set a notification value without overwriting a previous value
  - Overwrite a notification value
  - Set one or more bits in notification value
  - Increment notification value
- Can be used as binary/counting semaphore, event group or mailbox
- Limitations
  - 1 task per notification value
  - A sending task does not block to complete sending
- configUSE_TASK_NOTIFICATIONS must be 1 to use

Example: Direct to Task Notification as Wait-Signal

```
/* Task  will be notified when transmission complete */
static TaskHandle_t xTaskToNotify = NULL;

/* IO driver's transmit function. */
void StartTransmission(uint8_t *pcData, size_t xDataLength)
{
    /* xTaskToNotify should be NULL as no transmission
    is in progress */
    configASSERT(xTaskToNotify == NULL);

    xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* An interrupt is generated when transmission complete */
    vStartTransmit(pcData, xDatalength);
}
```


Example: Direct to Task Notification as Wait-Signal cont'd

```
/* The transmit end interrupt. */
void vTransmitEndISR( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Transmission was in progress. */
    configASSERT( xTaskToNotify != NULL );

    vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

    xTaskToNotify = NULL;

    /* If xHigherPriorityTaskWoken is pdTRUE then
        a context switch is needed */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```


Example: Direct to Task Notification as Wait-Signal cont'd

```
void vAFunctionCalledFromATask(uint8_t ucDataToTransmit, size_t xDataLength)
{
    uint32_t ulNotificationValue;
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS(200);

    StartTransmission(ucDataToTransmit, xDataLength);

    ulNotificationValue = ulTaskNotifyTake(pdTRUE,  xMaxBlockTime);

    if(ulNotificationValue == 1)
    {
        /* The transmission ended as expected. */
    }
    else
    {
        /* The call to ulTaskNotifyTake() timed out. */
    }
}
```


Example: Direct to Task Notification as Producer-Consumer

```
/* An ISR used as producer */
void vANInterruptHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken;

    prvClearInterruptSource();
    xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(xHandlingTask, &xHigherPriorityTaskWoken);
    /* Force a context switch if xHigherPriorityTaskWoken is pdTRUE */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```


Example: Direct to Task Notification as Producer-Consumer cont'd

```
/* A task used as a consumer */
void vHandlingTask( void *pvParameters )
{
    BaseType_t xEvent;
    const TickType_t xBlockTime = pdMS_TO_TICS(500);
    uint32_t ulNotifiedValue;

    for(;;)
    {
        ulNotifiedValue = ulTaskNotifyTake(pdFALSE, xBlockTime);
        if( ulNotifiedValue > 0 )
        {
            /* Perform any processing necessitated by the interrupt. */
        }
        else
        {
            /* Did not receive a notification within the expected time. */
        }
    }
}
```


Example: Direct to Task Notification as Event Group

```
/* Bits represent each interrupt source. */
#define TX_BIT    0x01
#define RX_BIT    0x02

/* Task that will receive notifications from ISRs */
static TaskHandle_t xHandlingTask;

/* Transmit interrupt service routine. */
void vTxISR(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    prvClearInterrupt();
    xTaskNotifyFromISR(xHandlingTask, TX_BIT, eSetBits,
                       &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```


Example: Direct to Task Notification as Event Group cont'd

```
/* Receive interrupt service routine */
void vRxISR(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    prvClearInterrupt();
    xTaskNotifyFromISR(xHandlingTask, RX_BIT, eSetBits,
                       &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```


Example: Direct to Task Notification as Event Group cont'd

```
/* Task notified by ISRs */
static void prvHandlingTask(void *pvParameter)
{
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS(500);
    BaseType_t xResult;

    for(;;)
    {
        xResult = xTaskNotifyWait(pdFALSE, ULONG_MAX, &ulNotifiedValue,
            xMaxBlockTime );

        if(xResult == pdPASS)
        {
            /* A notification was received */
            if((ulNotifiedValue & TX_BIT) != 0)
            {
                /* The TX ISR has set a bit. */
            }
            if(( ulNotifiedValue & RX_BIT) != 0)
            {
                /* The RX ISR has set a bit. */
            }
        }
    }
}
```


FreeRTOS Direct to Task Notification APIs

Creation	Control	Utilities
	xTaskNotify xTaskNotifyAndQuery xTaskNotifyGive xTaskNotifyWait ulTaskNotifyTake	xTaskNotifyStateClear

Notifying a Task, xTaskNotify

```
BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify,
    uint32_t ulValue, eNotifyAction eAction );
```

- ulValue is used to update notification value depending on eAction parameter
- eAction is action to perform when notifying a task

eNoAction	ulValue is ignored
eSetBits	notification value |= ulValue
eIncrement	notification value ++
eSetValueWithOverwrite	notification value = ulValue
eSetValueWithoutOverwrite	If notified task not pending on value update

- Return pdFAIL if eAction = eSetValueWithoutOverwrite and notification value is not updated, pdPASS otherwise

Notifying a Task

```
BaseType_t xTaskNotifyAndQuery(TaskHandle_t xTaskToNotify,
    uint32_t ulValue, eNotifyAction eAction,
    uint32_t *pulPreviousNotifyValue);
```

- xTaskNotifyAndQuery = xTaskNotify() + the notified task's previous notification value is returned
- pulPreviousNotifyValue is used to pass out the subject task's notification value

```
BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
```

- xTaskNotifyGive() = xTaskNotify() with the eAction = eIncrement
- Always return pdPASS

Wait for a Notify, xTaskNotifyWait

```
BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry,
    uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue,
    TickType_t xTicksToWait);
```

- ulBitsToClearOnEntry is used as a mask to clear some bits in the calling task's notification value on entry to xTaskskNotifyWait()
- ulBitsToClearOnExit is used as a mask to clear some bits in the calling task's notification value before xTaskskNotifyWait() exit if notification received
- pulNotificationValue pass out the task's notification value before any bits were cleared due to ulBitsToClearOnExit
- Return pdFALSE if timed out, pdTRUE otherwise

Wait for a Notify, ulTaskNotifyTake

```
uint32_t ulTaskNotifyTake(BaseType_t xClearCountOnExit,
    TickType_t xTicksToWait);
```

- ulTaskNotifyTake() = xSemaphoreTake() but for notification value
- If xClearCountOnExit is pdFALSE, decrement notification value on exit
- If xClearCountOnExit is pdTRUE, clear notification value on exit
- Return task's notification value before it is decremented or cleared

Clearing a Notify State, xTaskNotifyStateClear

```
BaseType_t xTaskNotifyStateClear(TaskHandle_t xTask);
```

- The notification itself has a state (pending or not pending)
  - This is different from task state
  - Not pending notification = Notification sent while a receiving task waiting for it
  - Pending notification = Notification sent while receiving task not waiting for it
- Pending notification is cleared when receiving task reads its notification value
- xTaskNotifyStateClear() clear a pending notification w/o receiving task has to read its notification value
- If xTask is NULL, the task clears its notification state
- Return pdPASS if cleared, pdFAIL otherwise

Example: Clearing a Notify State

```
/* An example UART send function that starts transmission then
    waits to be notified of transmission completed from ISR   */
void vSerialPutString(const signed char * const pcStringToSend,
                      unsigned short usStringLength )
{
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS(5000);
    /* xSendingTask holds the handle of the task waiting for the transmissio
       to complete */
    if((xSendingTask == NULL) && (usStringLength > 0))
    {
        /* Ensure calling task's notification state is not pending. */
        xTaskNotifyStateClear( NULL );
        /* Store the handle of the transmitting task. */
        xSendingTask = xTaskGetCurrentTaskHandle();
        UARTSendString(pcStringToSend, usStringLength);
        ulTaskNotifyTake(pdTRUE, xMaxBlockTime);
    }
}
```


Exercise: Direct to Task Notification

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- **SW Timers**
- Memory Management
- Interrupt Management
- Miscellaneous Topics

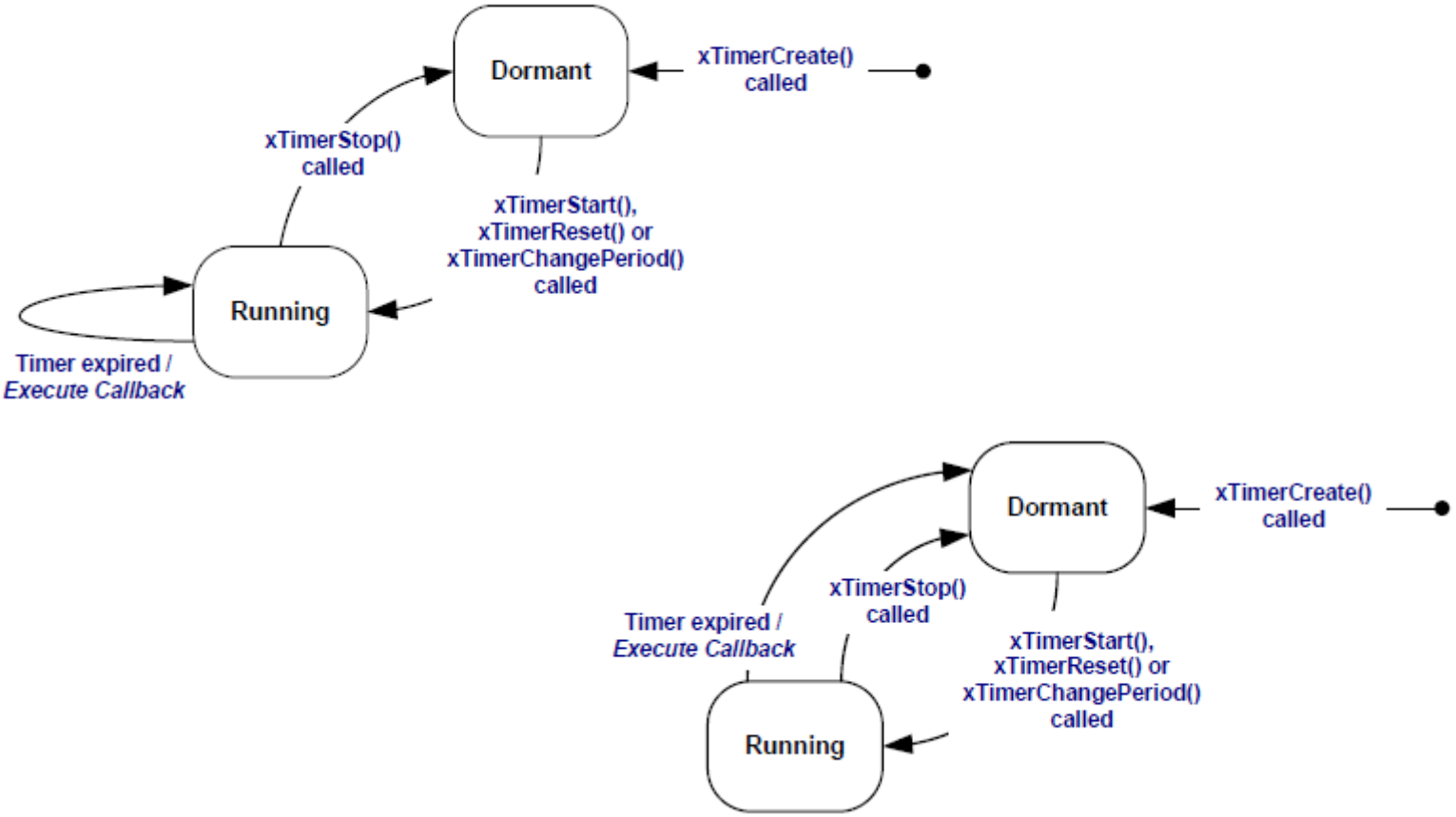
SW Timers vs. HW Timers

	SW Timer	HW Timer
Accuracy	Low	High
Precision	Low	High
Flexibility	High	Low
Limited by	SW or no limit	HW
Calls @ expiry	Callback	ISR

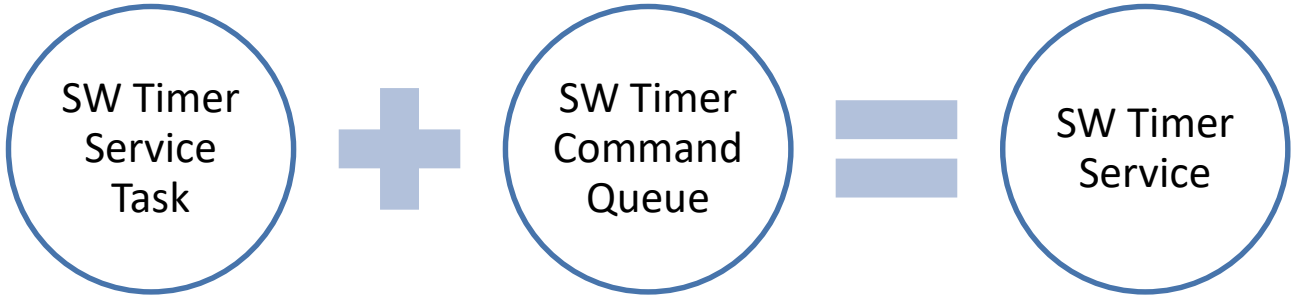
SW Timers

- Do not require HW support
- Use CPU only when a callback is running
- Controlled by Timer service task
- Callbacks run in timer service task and should not block
- configUSE_TIMERS must be 1 to use

Types of SW Timers



SW Timers Implementation



- configTIMER_TASK_PRIORITY, configTIMER_TASK_STACK_DEPTH and configTIMER_QUEUE_LENGTH must be set wisely
  - Priority affects responsiveness and accuracy
  - Stack depth constraints callbacks
  - Queue may fill up quickly
- Expiry time is calculated from time when command was sent to the timer command queue, not from reception
- Time commands are timestamped

FreeRTOS SW Timers APIs

Creation	Control	Utilities
xTimerCreate xTimerCreateStatic xTimerDelete	vTimerSetReloadMode xTimerStart xTimerStop xTimerChangePeriod xTimerReset xTimerPendFunctionCall	xTimerIsTimerActive pcTimerGetName pvTimerGetTimerID vTimerSetTimerID xTimerGetTimerDaemonTaskHandle xTimerGetPeriod xTimerGetExpiryTime

Creating a SW Timer, xTimerCreate

```
TimerHandle_t xTimerCreate(const char *pcTimerName,
    const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload,
    void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction );
```

- configSUPPORT_DYNAMIC_ALLOCATION must be 1 or undefined
- uxAutoReload if pdTRUE then timer is autoreload, one-shot otherwise
- pvTimerID is ID that is assigned to the timer being created
- pxCallbackFunction must have the following prototype

```
void vCallbackFunction(TimerHandle_t xTimer);
```

- Return SW timer handle or NULL

Creating/Deleting a SW Timer

```
TimerHandle_t xTimerCreateStatic(const char *pcTimerName,
    const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload,
    void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction
    StaticTimer_t *pxTimerBuffer );
```

- configSUPPORT_STATIC_ALLOCATION must be 1
- pxTimerBuffer will be used to hold SW timer data
- xTicksToWait is timeout for delete command to be successfully sent to timer command queue
- Return pdPASS if command sent, pdFAIL otherwise

```
BaseType_t xTimerDelete(TimerHandle_t xTimer, TickType_t xBlockTime );
```


Controlling a SW Timer

```
BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);
BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);
BaseType_t xTimerReset(TimerHandle_t xTimer, TickType_t xTicksToWait);
```

- xTimerStart() starts a SW timer if not running, if running xTimerStart() == xTimerReset()
- xTimerReset() resets starts a SW timer if running, if not running xTimerReset() == xTimerStart()
- xTimerStop() works on running timers only

Changing SW Timer Settings

```
void vTimerSetReloadMode(TimerHandle_t xTimer,
                         const UBaseType_t uxAutoReload);
BaseType_t xTimerChangePeriod(TimerHandle_t xTimer,
    TickType_t xNewPeriod, TickType_t xTicksToWait);
```

- xTimerChnagePeriod() changes period of SW timer if running, if not running xTimerChnagePeriod() == xTimerStart()

```
BaseType_t xTimerPendFunctionCall(PendedFunction_t xFunctionToPend,
    void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait);
```

- INCLUDE_xTimerPendFunctionCall must be 1
- Defer execution of xFuntionToPend to timer service task
- xFuntionToPend should have the following prototype

```
void vPendableFunction(void *pvParameter1, uint32_t ulParameter2);
```


Other SW Timer APIs

```
BaseType_t xTimerIsTimerActive(TimerHandle_t xTimer);

void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID);

void *pvTimerGetTimerID(TimerHandle_t xTimer);

const char * pcTimerGetName(TimerHandle_t xTimer);

TaskHandle_t xTimerGetTimerDaemonTaskHandle(void);

TickType_t xTimerGetPeriod(TimerHandle_t xTimer);

TickType_t xTimerGetExpiryTime(TimerHandle_t xTimer);
```

- Return next expiry time if timer is running
- If returned expiry ti,e < tick count, it will wait till tick count overflow

Example: Creating a SW Timer

```
#define NUM_TIMERS 5
TimerHandle_t xTimers[ NUM_TIMERS ];

/* Define a callback function that count the number of timer
expires, and stop the timer after 10 expiry.  The count is saved as
the ID */
void vTimerCallback(TimerHandle_t xTimer)
{
    uint32_t ulCount;

    configASSERT(xTimer);
    ulCount = (uint32_t) pvTimerGetTimerID(xTimer);
    ulCount++;
    if(ulCount >= 10)
    {
        /* A timer callback should not block! */
        xTimerStop(pxTimer, 0);
    }
    else
    {
       vTimerSetTimerID(xTimer, (void *) ulCount);
    }
}
```


Example: Creating a SW Timer cont'd

```
void main(void)
{
    long x;

    /* Timers will start running after scheduler starts. */
    for(x = 0; x < NUM_TIMERS; x++)
    {
        xTimers[x] = xTimerCreate("Timer",( 100 * x ) + 100,pdTRUE,(void *) 0,
                                  vTimerCallback);
        if(xTimers[x] == NULL)
        {
            /* The timer was not created. */
        }
        else
        {
            xTimerStart(xTimers[x], 0);
        }
    }
    ...
}
```


Exercise: SW Timers

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- **Memory Management**
- Interrupt Management
- Miscellaneous Topics

Memory Allocation

Dynamic Memory

- Fewer creation parameters
- Automatic
- Run-time failures
- RAM reusage is possible
- RTOS APIs to query heap usage
- 5 schemes

Static Memory

- More control w/ programmer
- Objects @ specific memory locations
- Compile-time failures
- Worst-case allocation

Dynamic Memory Allocation Options

Standard C malloc/free

- Not always available
- Large in footprint
- Not thread safe
- Not deterministic
- Suffer from external fragmentation
- Can complicate linking configuration
- Hard to debug for problems

Memory Pools

- Predetermined at compile time
- Most common in constrained RAM environment
- Deterministic
- Suffer from internal fragmentation
- Was not easy for users

FreeRTOS Dynamic Memory Manager

- Has 2 interfaces pvPortMalloc() and vPortFree()
- Can be used from application code
- FreeRTOS has 5 schemas for them (heapx.c)
- You can define your own extra schema and use it along w/ FreeRTOS built-in schemas
  - FreeRTOS schema for its code
  - Your schema for application code

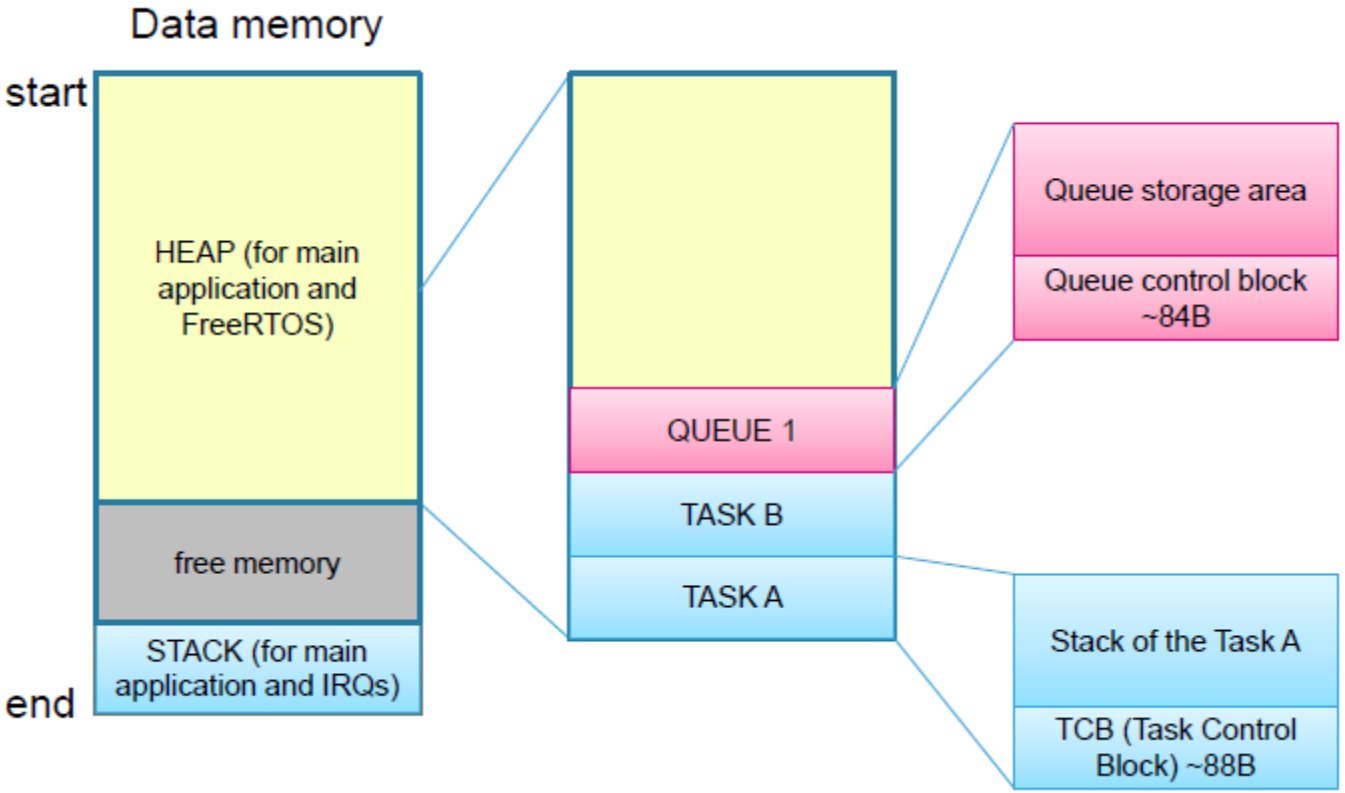
Configuring FreeRTOS Heap

- By default, heap is declared by FreeRTOS and placed in memory by linker
- Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows heap to be declared by application writer

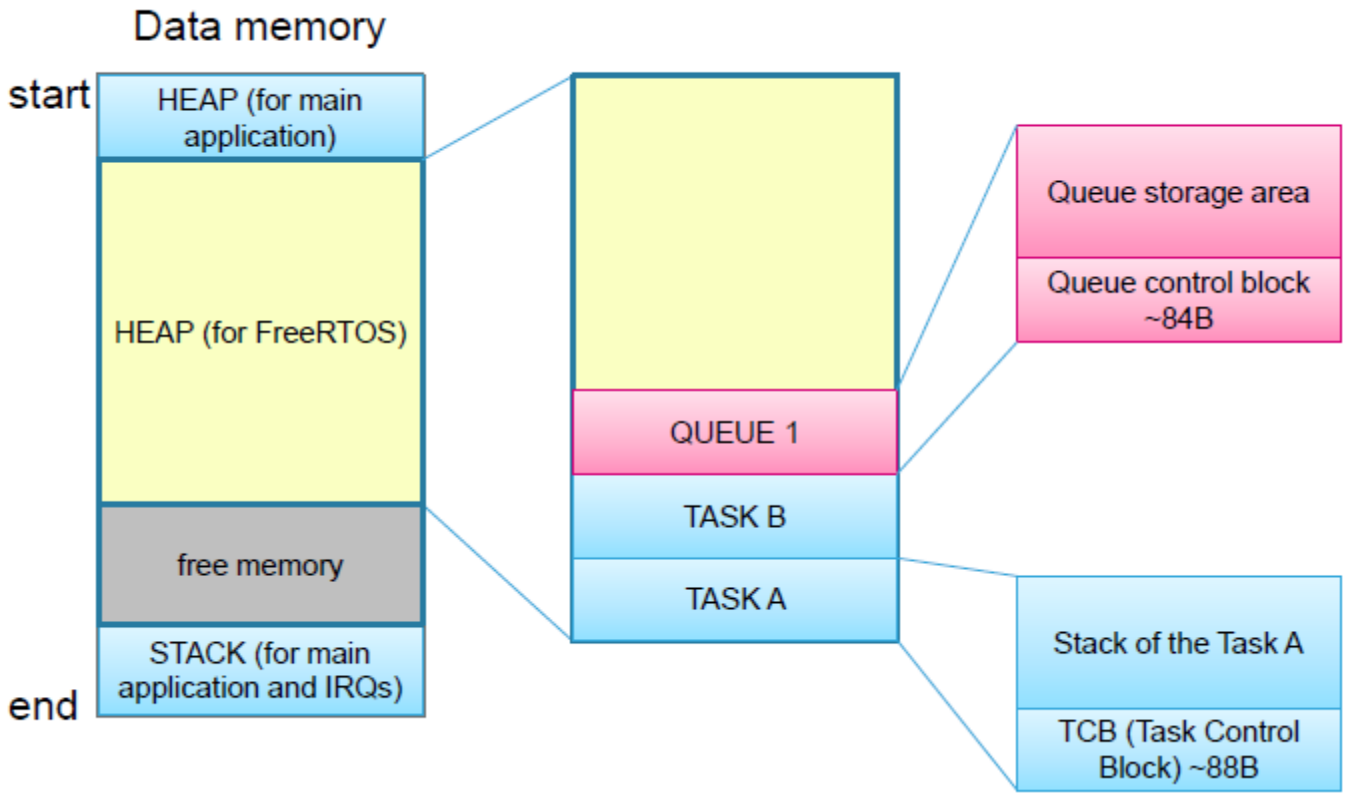
```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE];
```

- Must be defined if heap_1.c, heap_2.c or heap_4.c is used static in these files
- If configAPPLICATION_ALLOCATED_HEAP is 1, they should not be static and location should be defined
- Will be used as FreeRTOS heap

Stack and Heap Layouts – heap_3.c

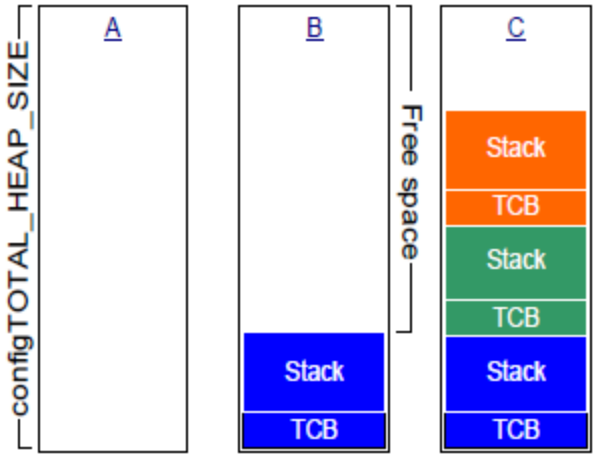


Stack and Heap Layouts - Others



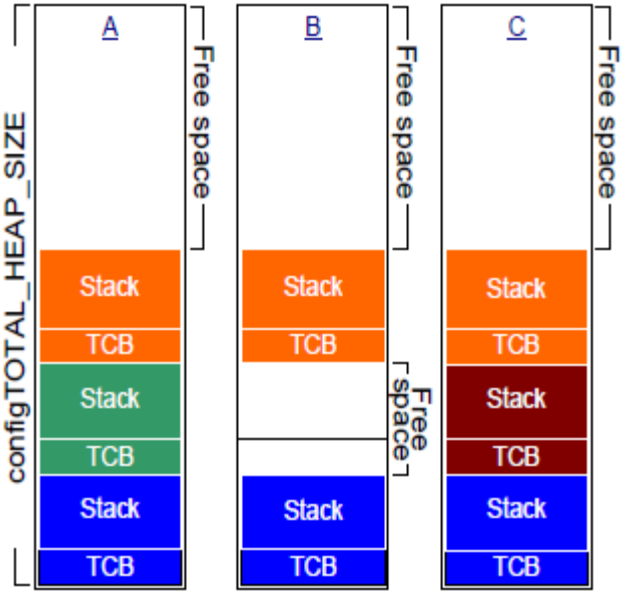
Heap_1.c

- Uses first fit algorithm
- Simplest/deterministic
- No freeing allowed
- Used in safety-critical systems



Heap_2.c

- Not recommended; kept for compatibility
- Uses best fit algorithm
- Not deterministic
- Freeing allowed
- No defragmentation
- Faster than malloc/free

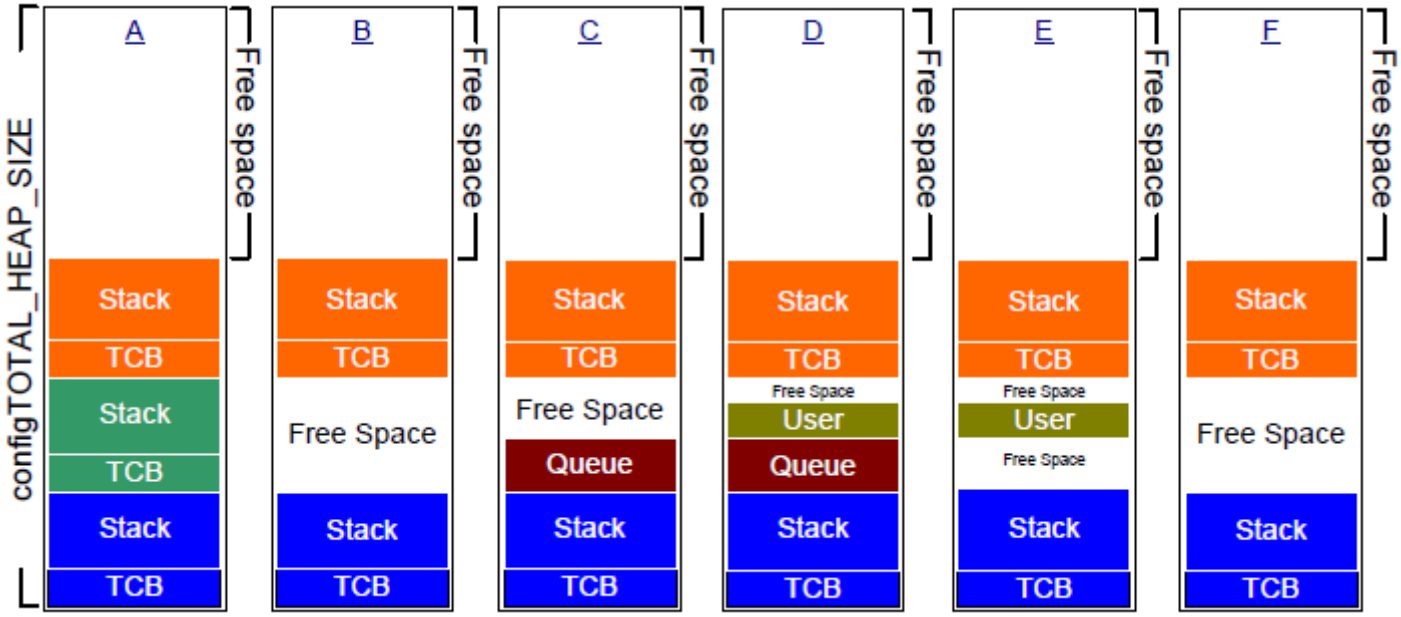


Heap_3.c

- Wraps malloc() and free() to make them thread-safe by suspending scheduler while in use
- Heap size defined by linker configuration
- Not deterministic.
- Will increase FreeRTOS size

Heap_4.c

- = Heap_2.c + defragmentation
- Not deterministic but more efficient than malloc()/free()



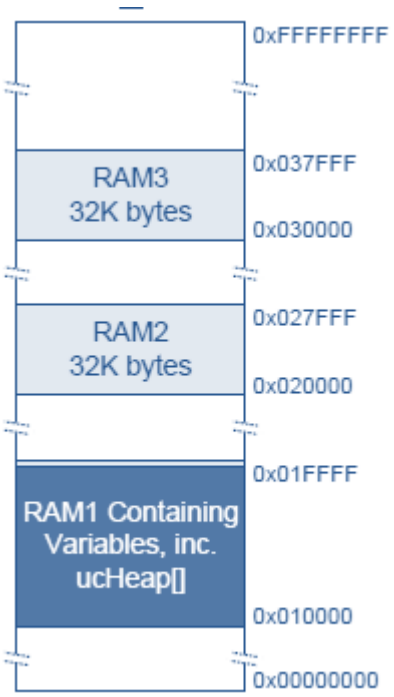
Heap_5.c

- = Heap_4.c + non-contiguous heap
- Heap_5 is initialized using vPortDefineHeapRegions()

```
void vPortDefineHeapRegions(const HeapRegion_t * const pxHeapRegions);

typedef struct HeapRegion
{
    /* The start address of a RAM block */
    uint8_t *pucStartAddress;
    /* The size of the block */
    size_t xSizeInBytes;
} HeapRegion_t;
```


Example: vPortDefineHeapRegions

```
/* Define the start address and size of 2  RAM regions not used by the
linker. */
#define RAM2_START_ADDRESS ((uint8_t*)0x00020000)
#define RAM2_SIZE (32*1024)
#define RAM3_START_ADDRESS ((uint8_t*)0x00030000)
#define RAM3_SIZE (32*1024)

/* Declare an array that will be part of the heap used by heap_5.
    The array will be placed in RAM1 by the linker. */
#define RAM1_HEAP_SIZE (30*1024)
static uint8_t ucHeap[RAM1_HEAP_SIZE];

/* Create HeapRegion_t definitions */
const HeapRegion_t xHeapRegions[] =
{
    {ucHeap, RAM1_HEAP_SIZE},
    {RAM2_START_ADDRESS, RAM2_SIZE},
    {RAM3_START_ADDRESS, RAM3_SIZE},
    {NULL, 0} /* Marks the end of the array. */
};

int main(void)
{
    /* Initialize heap_5. */
    vPortDefineHeapRegions( xHeapRegions );
    /* Add application code here. */
}
```



Heap Related FreeRTOS APIs

```
size_t xPortGetFreeHeapSize(void);
```

- Not available w/ heap_3
- Return free bytes in heap @ call

```
size_t xPortGetMinimumEverFreeHeapSize(void);
```

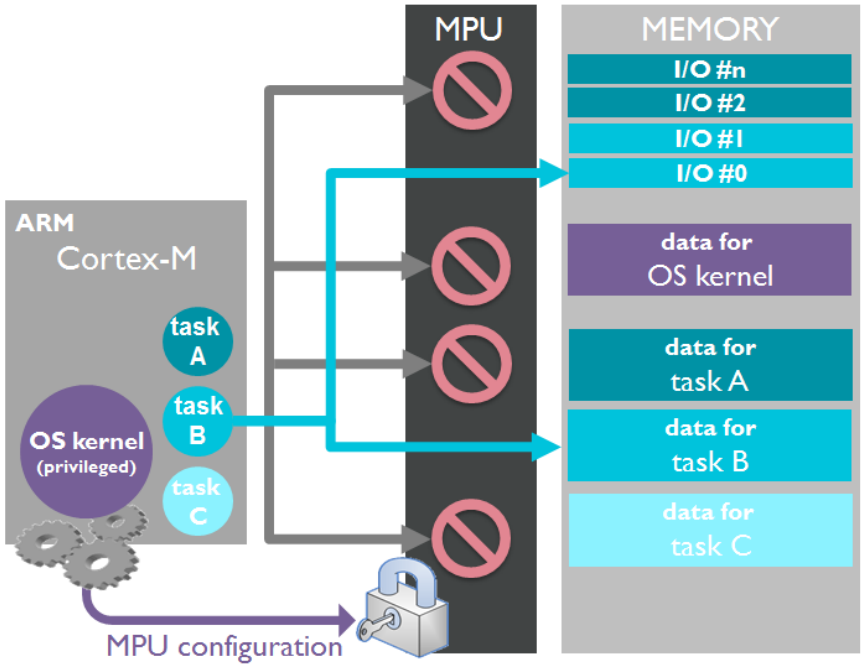
- Return minimum # of unallocated bytes that have existed

```
void vApplicationMallocFailedHook(void);
```

- configUSE_MALLOC_FAILED_HOOK must be 1
- called when pvPortMalloc() fails

Memory Protection Unit

- For ARM CortexM, there are 2 ports standard FreeRTOS port and FreeRTOS-MPU
- FreeRTOS-MPU will protect
  - Kernel from invalid execution by tasks
  - Data used by kernel from tasks
  - Configuration of core resources
  - IO devices and memory
- FreeRTOS-MPU will guarantee
  - Guarantee task stack overflows are detection
  - Guarantee task isolation



FreeRTOS-MPU Features

- Compatible w/ ARM Cortex-M3/M4
- Tasks can run in either Privileged mode or User mode
  - In FreeRTOS port, all tasks run in Privileged mode
- User mode tasks
  - Only access their own stack + up to 3 user definable memory regions
  - Can pass messages using queue and shared memory regions are discouraged
- User definable memory regions
  - Assigned @creation and can be reconfigured later
  - Parameterized individually

FreeRTOS-MPU Features cont'd

- Privileged mode task can set itself into User mode but cannot set itself back
- FreeRTOS API is in Flash region that can only be accessed while controller is in Privileged mode (Calling API cause mode switch)
- FreeRTOS data is in a region of RAM that can only be accessed while controller is in Privileged mode.
- System peripherals can be accessed while controller is in Privileged mode
- Standard peripherals are accessible by any code but can be protected using a user definable memory region

Creating a Protected Task

- Tasks not using MPU are created using xTaskCreate() and can run in either Privileged or User modes
- MPU aware tasks are created using called xTaskCreateRestricted()

Defining an MPU Region

- 8 regions (0 to 7) max @ a time
- Overlapping regions follow rules of region w/ higher number
- Regions 0 to 4 are used by kernel
- Regions 5 to 7 are used by user task and are reconfigured during context switch
- Region size must be a power of 2 starting 32 bytes and up to 64 G Bytes
- Regions start address must be a multiple of the region size

FreeRTOS-MPU Linker Configuration

- Require 2 segments one for kernel functions and other for kernel data

- Require 8 variables
  - FLASH start/end
  - Kernel functions start/end
  - SRAM start/end
  - Kernel data start/end

Accessing Data from a User Mode Task

- If usser mode task needs value of a globally declared variable, value must be copied into a variable that is on the task stack
- Initially, create the task in Privileged mode, and then copy value into a stack variable, before switching to User mode
- Pass value into the task using the task parameter

Creating a Protected Task, xTaskCreateRestricted

```
BaseType_t xTaskCreateRestricted(TaskParameters_t *pxTaskDefinition,
        TaskHandle_t *pxCreatedTask);
```

```
typedef struct xTASK_PARAMETERS
{
    TaskFunction_t pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    UBaseType_t uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    MemoryRegion_t xRegions[portNUM_CONFIGURABLE_REGIONS];
}TaskParameters_t;

typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
}MemoryRegion_t;
```

Creating a Protected Task, xTaskCreateRestricted cont'd

- puxStackBuffer is set to address of statically declared stack
- If puxStackBuffer set to NULL, vPortMallocAligned() will be called to allocate stack
- ulParameters defines how the task is permitted to access the memory region and can take bitwise OR of

Attribute	Privileged Task	User Task
portMPU_REGION_READ_WRITE	Full Access	Full Access
portMPU_REGION_PRIVILEGED_READ_ONLY	Read Only	No Access
portMPU_REGION_READ_ONLY	Read Only	Read Only
portMPU_REGION_PRIVILEGED_READ_WRITE	Full Access	No Access
portMPU_REGION_EXECUTE_NEVER	No Execute	No Execute

Example: Creating a Protected Task

```
static portSTACK_TYPE xTaskStack[128] __attribute__((aligned(128*4)));
char cReadOnlyArray[ 512 ] __attribute__((aligned(512)));
static const TaskParameters_t xTaskDefinition =
{
    vTaskFunction,
    "A task",
    128,
    NULL,
    1,
    {
        /* Base address   Length                      Parameters */
        { cReadOnlyArray, mainREAD_ONLY_ALIGN_SIZE, portMPU_REGION_READ_ONLY },
        { 0,              0,                        0                          },
        { 0,              0,                        0                          },
    }
};

void main( void )
{
    xTaskCreateRestricted( &xTaskDefinition, NULL );

    ...
}
```


Other FreeRTOS-MPU APIs

```
void vTaskAllocateMPURegions(TaskHandle_t xTaskToModify,
                const MemoryRegion_t * const xRegions);


void portSWITCH_TO_USER_MODE(void);
```


Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- **Interrupt Management**
- Miscellaneous Topics

FreeRTOS Interrupt Safe API's

- Calling RTOS APIs from ISR must be done w/ care
- FreeRTOS has define a set of APIs to be used from ISR ("FromISR" appended to their name)
- Benefits of FromISR APIs
  - Efficiency in terms of footprint
  - More cohesive design
  - Simpler FreeRTOS porting
- Disadvantage is when designing a non FreeRTOS API that should be called from ISR and Rask and it should use a FreeRTOS APIs

FreeRTOS Interrupt Safe API's cont'd

- xHigherPriorityTaskWoken Parameter
  - Define it inside your ISR and initialize it to pdFALSE
  - Passed to "FromISR" API and is set to pdTRUE if the API readies a blocked task and a context switch is needed
  - Its usage is optional. If not required, then set it to NULL

- Context switching can not be done within ISR

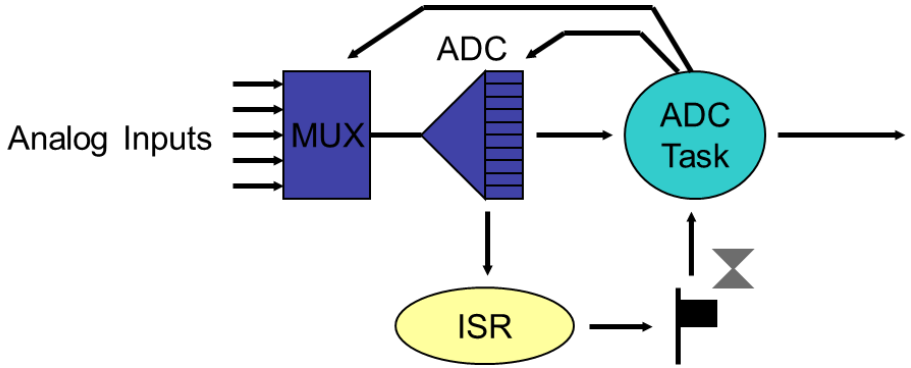
- portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() should be called @ ISR end to requext context switching
  - One of them or both are defined in your port

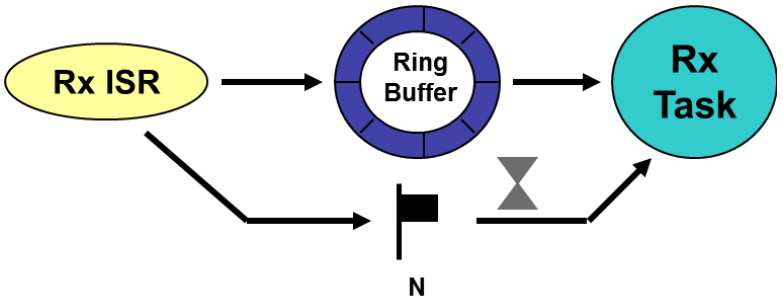
```
portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
```

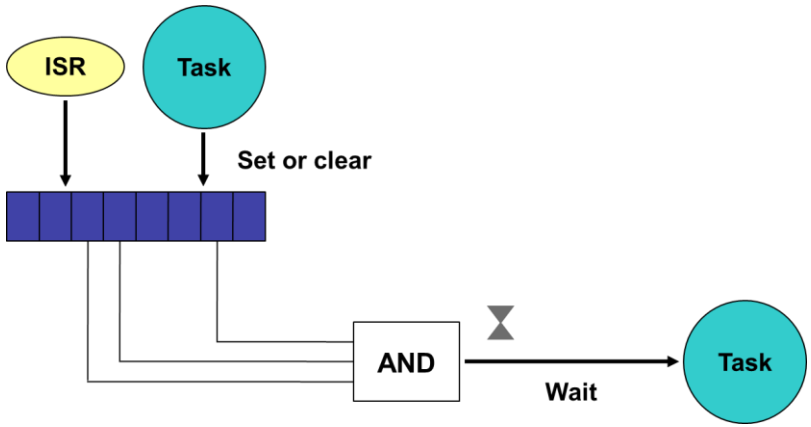

Example: ISR

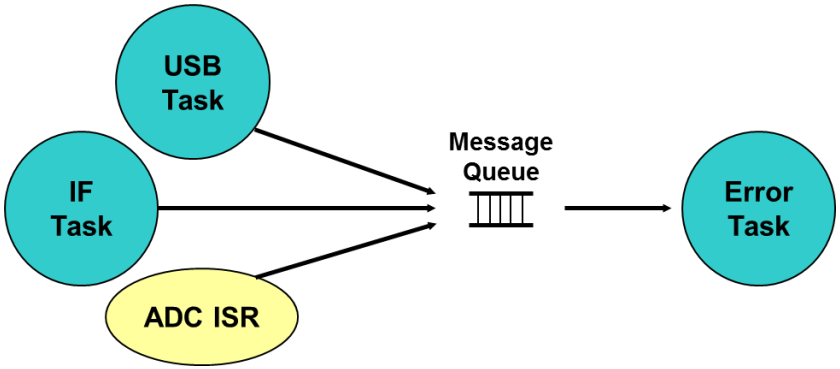
```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    xSemaphoreGiveFromISR(xBinarySemaphore, &xHigherPriorityTaskWoken);

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```


Decentralized Deferred Interrupt Handling









Centralized Deferred Interrupt Handling

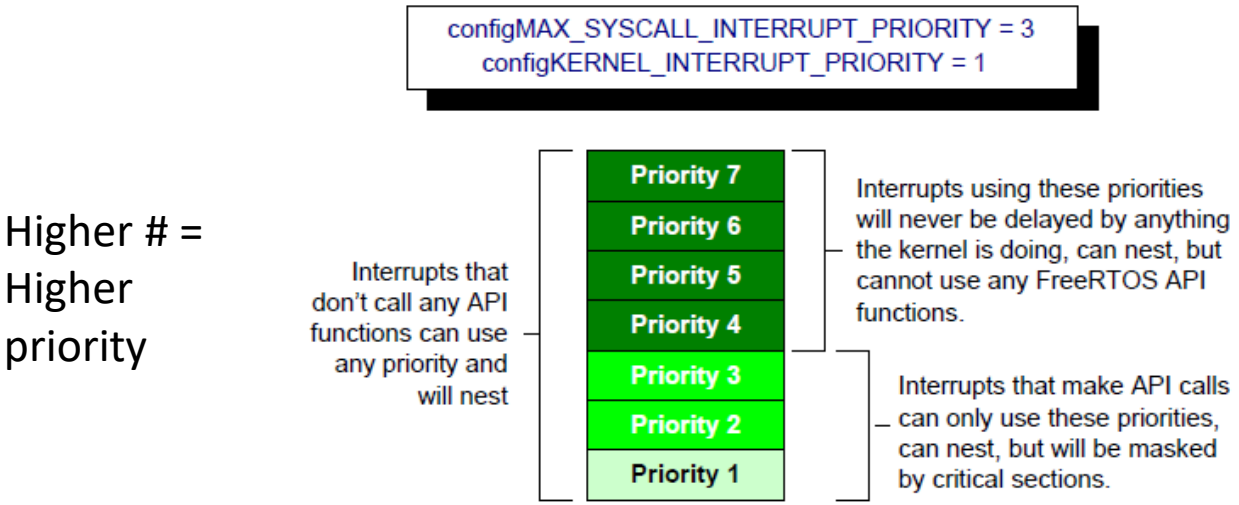
- Can be done by xTimerPendFunctionCallFromISR() to defer handling to the SW timer service task

Decentralized	Centralized
Advantages: • Reduced latency • Greater flexibility	Advantages: • Less consumption • Simplified model
Disadvantages: • Greater consumption	Disadvantages: • Less flexibility • Less determinism

Interrupt Nesting

- Ports support ISR nesting must define the following

configMAX_SYSCALL_INTERRUPT_PRIORITY or configMAX_API_CALL_INTERRUPT_PRIORITY	Highest that can call FromISR APIs
configKERNEL_INTERRUPT_PRIORITY	Tick ISR priority



Interrupt Nesting cont'd

- configKERNEL_INTERRUPT_PRIORITY should be set to the lowest priority
- For ports that only implement configKERNEL_INTERRUPT_PRIORITY, this is considered configMAX_SYSCALL_INTERRUPT_PRIORITY/configMAX_API_CALL_INTERRUPT_PRIORITY
- Interrupt nesting model is achieved if configMAX_SYSCALL_INTERRUPT_PRIORITY ≥ configKRNEL_INTERRUPT_PRIORITY.

Exercise: Interrupt Management

Outline

- Introduction
- FreeRTOS Overview
- RTOS Multitasking
- Inter-task Access Synchronization
- Inter-task Event Synchronization
- Inter-task Communication
- Direct to Task Notification
- SW Timers
- Memory Management
- Interrupt Management
- **Miscellaneous Topics**

configASSERT()

- The C assert() is not available on all compilers
- FreeRTOS defines its own, configASSERT() which is empty by default
- Can be defined by application writer in FreeRTOSConfig.h
- Greatly assist in run-time debugging/productivity on expense of code size

```
void vAssertCalled(const char *pcFile, uint32_t ulLine)
{
    RecordErrorInformationHere(pcFile, ulLine);
    taskDISABLE_INTERRUPTS();
    for(;;);
}

#define configASSERT( x) if((x) == 0) vAssertCalled(__FILE__, __LINE__)
```


Hooks

- Extension points in RTOS code left to the programmer
- There are debug-related hooks and trace-related hooks
- Debug-related hooks
  - Idle task hook
  - Tick hook enabled by configUSE_TICK_HOOK and could be used to implement SW timer functionality

```
void vApplicationTickHook(void);
```

  - Malloc failed hook
  - Stack overflow hook
  - Daemon task startup hook called from SW timer service task after startup (post scheduler initialization)

```
void vApplicationDaemonTaskStartupHook(void);
```


Trace Hook Macros

- Permit data collection on how application is behaving
- Placed @ key point of interest inside FreeRTOS code
- Redefined based on interest @ end of FreeRTOSConfig.h
- Be careful when you implement them
- Used by FreeRTOS aware debuggers, dynamic analysis tools or IDEs
- List is in here: https://www.freertos.org/rtos-trace-macros.html

Example: Trace Hook Macros

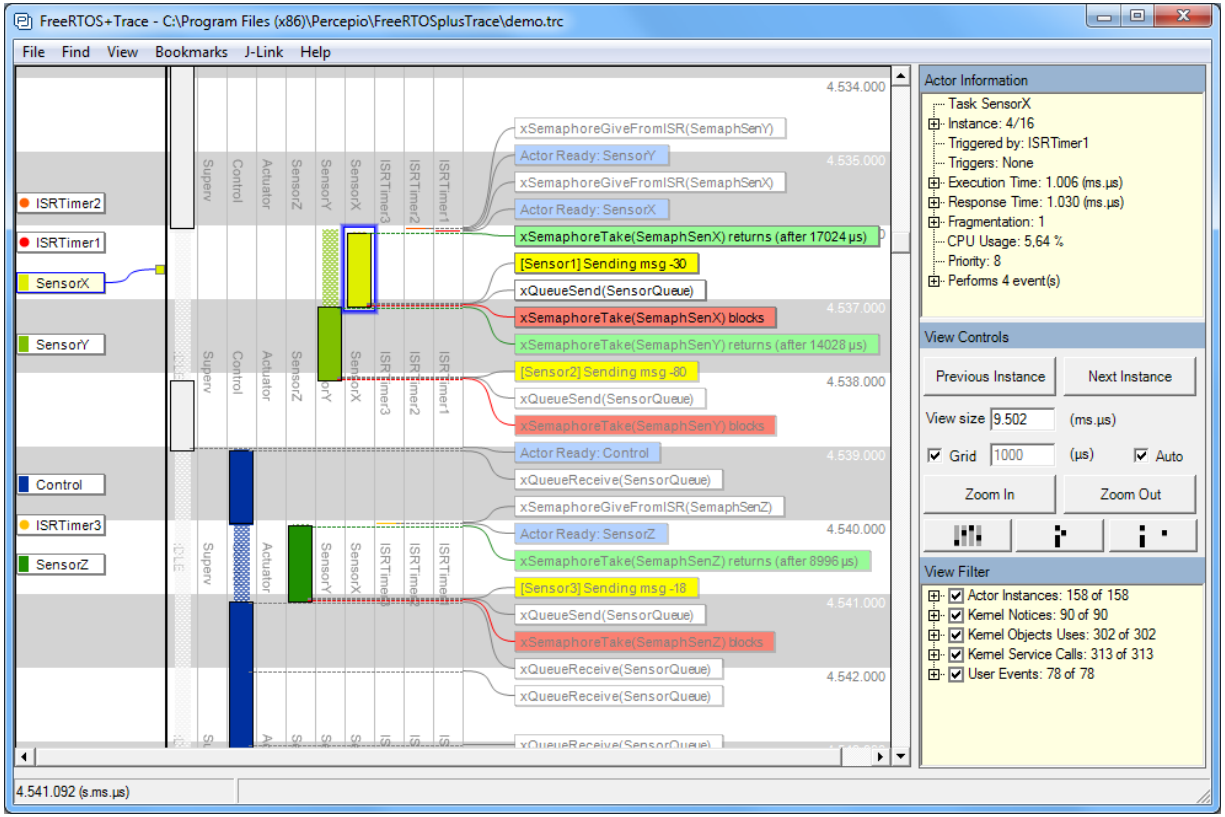
```
/* Define the traceTASK_SWITCHED_IN() macro to output the voltage associated
with the task being selected to run on port 0. */
#define traceTASK_SWITCHED_IN() vSetAnalogueOutput(0, (int)pxCurrentTCB->pxTaskTag)


/* traceBLOCKING_ON_QUEUE_RECEIVE() is just one of the macros that can be used to
record why a context switch is about to occur. */
#define traceBLOCKING_ON_QUEUE_RECEIVE(xQueue)    \
    ulSwitchReason = reasonBLOCKING_ON_QUEUE_READ;

/* log_event() is an application defined function that logs which tasks ran when,
and why. */
#define traceTASK_SWITCHED_OUT()    \
    log_event( pxCurrentTCB, ulSwitchReason );
```


TracealyzerTM

- From Percepio
- Powerful runtime analysis tool
- Captures dynamic behavior information for offline display
- Can obtain trace through your debugger or any communication interface
- Trace library use FreeRTOS trace macros

TracealyzerTM cont'd

- 25+ graphically Interconnected views



Power Saving

- Idle task hook can place controller into low power mode but limited by tick
- FreeRTOS tickless idle mode stops tick then makes tick count correcting adjustment when tick restarts
  - Deep power saving until interrupt occurs, or RTOS readies a blocked task

portSUPPRESS_TICKS_AND_SLEEP(xExpectedIdleTime)

- configUSE_TICKLESS_IDLE must be 1 (FreeRTOS implementation) or 2 (user implementation)
- Some ports may not have built-in implementation
- Called by FreeRTOS when Idle task is running and configEXPECTED_IDLE_TIME_BEFORE_SLEEP ticks happened

Example: portSUPPRESS_TICKS_AND_SLEEP

```
#define portSUPPRESS_TICKS_AND_SLEEP(xIdleTime) vApplicationSleep(xIdleTime)


void vApplicationSleep( TickType_t xExpectedIdleTime )
{
    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;
    eSleepModeStatus eSleepStatus;

    /* Read current time from a source that will remain operational
    while the controller is in a low power state. */
    ulLowPowerTimeBeforeSleep = ulGetExternalTime();
    prvStopTickInterruptTimer();
    disable_interrupts();
    /* Ensure it is still OK to enter the sleep mode. */
    eSleepStatus = eTaskConfirmSleepModeStatus();
    if(eSleepStatus == eAbortSleep)
    {
        /* A task is ready. Restart the tick and exit the critical section. */
        prvStartTickInterruptTimer();
        enable_interrupts();
    }
```


Example: portSUPPRESS_TICKS_AND_SLEEP cont'd

```
    else
    {
        if(eSleepStatus == eNoTasksWaitingTimeout)
        {
            /* Indefinite sleep, No need for wakeup */
            /* No SW timers and all tasks are blocked for infinite timeout */
            prvSleep();
        }
        else
        {
            /* Configure wakeup interrupt */
            vSetWakeTimeInterrupt( xExpectedIdleTime );
            /* Enter the low power state. */
            prvSleep();
            ulLowPowerTimeAfterSleep = ulGetExternalTime();
            /* Correct the kernels tick */
            vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );
        }
        enable_interrupts();
        /* Restart the timer that is generating the tick interrupt. */
        prvStartTickInterruptTimer();
    }
}
```


Power Saving in ARM Cortex-M

- SysTick timer to generate tick interrupts
- SysTick is clocked @ core speed
  - Limits maximum tickles period and recommended to use another tick source
- configPRE_SLEEP_PROCESSING(xExpectedIdleTime)/configPOST_SLEEP_PROCESSING(xExpectedIdleTime) can be used to make use of low power features during sleep

Method	Tick Timer	Configuration
Built-in	SysTick @ core speed	configUSE_TICKLESS_IDLE = 1
Built-in	SysTick < core speed	configUSE_TICKLESS_IDLE = 1 configSYSTICK_CLOCK_HZ = SysTick freqency
Built-in	Other tick source	
User-defined	Other tick souce	

Power Saving in ARM Cortex-M cont'd

Method	Tick Timer	Configuration
Built-in	SysTick @ core speed	configUSE_TICKLESS_IDLE = 1
Built-in	SysTick < core speed	configUSE_TICKLESS_IDLE = 1 configSYSTICK_CLOCK_HZ = SysTick freqency
Built-in	Other tick source	Invalid
User-defined	Other tick souce	configUSE_TICKLESS_IDLE = 2

Stack Overflow Detection

- Do not forget uxTaskGetStackHighWaterMark()
- 2 methods of detection on flat memory architectures
- configCHECK_FOR_STACK_OVERFLOW select method (1 or 2)
- Checking introduces a context switch overhead

Method 1	Method 2
Quicker May miss overflows Less accurate	Used w/ Method 1 Stack is filled w/ key value @ creation Checks last 16 bytes Less efficient May miss overflows More accurate

Stack Overflow Hook

```
void vApplicationStackOverflowHook(TaskHandle_t xTask,
    signed char *pcTaskName);
```

- Called when an overflow is detected
- Depending on overflow severity, parameters may be corrupted
- Inspect pxCurrentTCB directly instead

Blocking on Multiple Objects

- Queue sets enables an RTOS task to block on multiple queues and/or semaphores at the same time

- FreeRTOS Queue Set APIs
  - configUSE_QUEUE_SETS must be 1

Creation	Control	Utilities
xQueueCreateSet	xQueueAddToSet xQueueRemoveFromSet xQueueSelectFromSet	

Creating a Queue Set, xQueueCreateSet

```
QueueSetHandle_t xQueueCreateSet(const UBaseType_t uxEventQueueLength);
```

- Blocking on a queue set that contains mutex will not cause mutex holder to inherit the priority of the blocked task
- A receive from queue or take from semaphore must not be performed on a queue set member unless xQueueSelectFromSet() has first returned a handle to that set member
- Return queue set handle or NULL

Creating a Queue Set, xQueueCreateSet cont'd

- uxEventQueueLength specifies maximum # of events that can be queued at once
- uxEventQueueLength must be large enough
  - Queue needs its queue length
  - Binary semaphore or mutex each need 1
  - Counting semaphore needs max count
- Return queue set handle or NULL

Adding/Removing from Queue Set

```
BaseType_t xQueueAddToSet(QueueSetMemberHandle_t xQueueOrSemaphore,
                          QueueSetHandle_t xQueueSet);
BaseType_t xQueueRemoveFromSet(QueueSetMemberHandle_t xQueueOrSemaphore,
                               QueueSetHandle_t xQueueSet);
```

- Queues and semaphores must be empty when added/removed and should be a member of 1 and only 1 queue set
- Return pdPASS if successful, pdFAIL otherwise

Selecting from Queue Set, xQueueSelectFromSet

```
QueueSetMemberHandle_t xQueueSelectFromSet(QueueSetHandle_t xQueueSet,
                const TickType_t xTicksToWait);
```

- Return NULL if timeout

Example: Queue Set

```
/* Define the lengths of the queues that will be added to the queue set. */
#define QUEUE_LENGTH_1      10
#define QUEUE_LENGTH_2      10

/* Binary semaphores have an effective length of 1. */
#define BINARY_SEMAPHORE_LENGTH 1

/* Define the size of the item to be held by queue 1 and queue 2 respectively.
The values used here are just for demonstration purposes. */
#define ITEM_SIZE_QUEUE_1   sizeof(uint32_t)
#define ITEM_SIZE_QUEUE_2   sizeof(something_else_t)

/* The combined length of the two queues and binary semaphore that will be
added to the queue set. */
#define COMBINED_LENGTH (QUEUE_LENGTH_1 +\
                         QUEUE_LENGTH_2 +\
                         BINARY_SEMAPHORE_LENGTH)
```


Example: Queue Set cont'd

```
void vAFunction( void )
{
    static QueueSetHandle_t xQueueSet;
    QueueHandle_t xQueue1, xQueue2, xSemaphore;
    QueueSetMemberHandle_t xActivatedMember;
    uint32_t xReceivedFromQueue1;
    something_else_t xReceivedFromQueue2;

    /* Create the queue set large enough to hold an event for every space in
    every queue and semaphore that is to be added to the set. */
    xQueueSet = xQueueCreateSet(COMBINED_LENGTH);

    /* Create the queues and semaphores that will be contained in the set. */
    xQueue1 = xQueueCreate(QUEUE_LENGTH_1, ITEM_SIZE_QUEUE_1);
    xQueue2 = xQueueCreate(QUEUE_LENGTH_2, ITEM_SIZE_QUEUE_2);

    /* Create the semaphore that is being added to the set. */
    xSemaphore = xSemaphoreCreateBinary();

    /* Check everything was created. */
    configASSERT(xQueueSet);
    configASSERT(xQueue1);
    configASSERT(xQueue2);
    configASSERT(xSemaphore);
```


Example: Queue Set cont'd

```
for(;;)
{
    /* Block to wait for something to be available from the queues or
    semaphore that have been added to the set.  Don't block longer than
    200ms. */
    xActivatedMember = xQueueSelectFromSet(xQueueSet,
                                           200/portTICK_PERIOD_MS);

    /* Which set member was selected?  Receives/takes can use a block time
    of zero as they are guaranteed to pass because xQueueSelectFromSet()
    would not have returned the handle unless something was available. */
    if(xActivatedMember == xQueue1)
    {
        xQueueReceive(xActivatedMember, &xReceivedFromQueue1, 0);
        vProcessValueFromQueue1(xReceivedFromQueue1);
    }
    else if(xActivatedMember == xQueue2)
    {
        xQueueReceive(xActivatedMember, &xReceivedFromQueue2, 0);
        vProcessValueFromQueue2(&xReceivedFromQueue2);
    }
    else if(xActivatedMember == xSemaphore)
    {
        /* Take the semaphore to make sure it can be "given" again. */
        xSemaphoreTake(xActivatedMember, 0);
        vProcessEventNotifiedBySemaphore();
        break;
    }
    else
    {
        /* The 200ms block time expired without an RTOS queue or semaphore
        being ready to process. */
    }
}
```


Use of printf() or sprintf()

- If provided may be not thread-safe or of large size

- printf-stdarg.c is included in many example projects
  - Defines a minimal and stack-efficient implementation of sprintf()
  - Defines mechanism for directing the printf() output to a port character by character
  - 3rd party contribution and licensed separately from FreeRTOS

Exercise: Common Sources of Error

Symptom

1. Adding a simple task crashes demo
2. Using API inside ISR causes crash
3. Sometimes ISR crashes
4. Scheduler crashes after its start
5. Interrupts left disabled
6. Critical sections do not nest
7. Application crash before scheduler start
8. Application crash during critical section or scheduler suspension

Solution

a. Increase heap size
b. Remove demo tasks
c. Only use FromISR APIs from ISR
d. Check ISR is not causing stack overflow
e. Check ISR is using correct compiler syntax
f. ISR priorities are correctly set
g. Put CPU in privileged mode before schedule start
h. Install ISRs correctly
i. Only use FreeRTOS APIs to manipulate interrupt state
j. Only create objects before scheduler start
k. Do not use call APIs depending on scheduler or ISRs

- To contact us:
  - www.swift-act.com
  - training@swift-act.com
  - (+2)0122-3600-207