OST
Ostschweizer
Fachhochschule

FreeRTOS_QuickReferenceChart v1_11

FreeRTOS V10.1.1

TASK MANAGEMENT	BaseType_t **xTaskCreate**(TaskFunction_t pvTaskCode, const char * const pcName, unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pvCreatedTask);		Create a new instance of a task
	void **vTaskDelete**(TaskHandle_t pxTask);		Delete an instance of a task
	void **vTaskSuspend**(TaskHandle_t pxTaskToSuspend);		Suspend a task
	void **vTaskSuspendAll**(void);		Suspend all tasks

	void **vTaskResume**(TaskHandle_t pxTaskToResume);
	BaseType_t **xTaskResumeFromISR**(TaskHandle_t pxTaskToResume);
		⚡	Resume a task

	BaseType_t **xTaskResumeAll**(void);		Resume all tasks
	UBaseType_t **uxTaskPriorityGet**(TaskHandle_t pxTask);		Get the priority of a task
	void **vTaskPrioritySet**(TaskHandle_t pxTask, UBaseType_t uxNewPriority);		Set the priority of a task
	void **vTaskDelay**(TickType_t xTicksToDelay);		Place calling task into blocked state for a fixed number of ticks
	void **vTaskDelayUntil**(TickType_t *pxPreviousWakeTime, TickType_t xTimeIncrement);		Place calling task into blocked state until the absolute time is reached
	TaskHandle_t **xTaskGetCurrentTaskHandle**(void);		Get the handle of the running task
	UBaseType_t **uxTaskGetNumberOfTasks**(void);		Get the number of all actual existing tasks
	eTaskState **eTaskGetState**(TaskHandle_t pxTask);		Get the actual state of a task
	UBaseType_t **uxTaskGetStackHighWaterMark**(TaskHandle_t xTask);		Get the minimum amount of remaining stack of a task
	char * **pcTaskGetTaskName**(TaskHandle_t xTaskToQuery);		Get the name of a task
	TaskHookFunction_t **xTaskGetApplicationTaskTag**(TaskHandle_t xTask);		Get the tag value of a task
	void **vTaskSetApplicationTaskTag**(TaskHandle_t xTask, TaskHookFunction_t pxTagValue);		Set a tag value associated to a task
	void **taskDISABLE_INTERRUPTS**(void);		Disable interrupts (priority level!)
	void **taskENABLE_INTERRUPTS**(void);		Enable all interrupt priorities
	void **taskENTER_CRITICAL**(void);		Disable interrupts (priority level!)
	void **taskEXIT_CRITICAL**(void);		Enable interrupts
	void **taskYIELD**(void);		Yield to another task of equal priority
SEMAPHORES	SemaphoreHandle_t **xSemaphoreCreateBinary**(void);		Create a binary semaphore
	SemaphoreHandle_t **xSemaphoreCreateCounting**(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);		Create a counting semaphore
	SemaphoreHandle_t **xSemaphoreCreateMutex**(void);		Create a mutex type semaphore
	SemaphoreHandle_t **xSemaphoreCreateRecursiveMutex**(void);		Create a recursive mutex type semaphore
	void **vSemaphoreDelete**(SemaphoreHandle_t xSemaphore);		Delete a semaphore
	BaseType_t **xSemaphoreGiveFromISR**(SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Releases a semaphore
	BaseType_t **xSemaphoreGiveRecursive**(SemaphoreHandle_t xMutex);		Releases a recursive mutex type semaphore

	BaseType_t **xSemaphoreTake**(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
	BaseType_t **xSemaphoreTakeFromISR**(SemaphoreHandle_t xSemaphore,
	BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Obtain a semaphore

	BaseType_t **xSemaphoreTakeRecursive**(SemaphoreHandle_t xMutex, TickType_t xTicksToWait);		Obtain a recursive mutex type semaphore
	TaskHandle_t **xSemaphoreGetMutexHolder**(SemaphoreHandle_t xMutex);		Return the handle of the holding task
QUEUE	void **vQueueAddToRegistry**(QueueHandle_t xQueue, char *pcQueueName);		Assign a name to a queue (Debug)
	QueueHandle_t **xQueueCreate**(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);		Creates a queue
	void **vQueueDelete**(TaskHandle_t pxQueueToDelete);		Delete a queue
	BaseType_t **xQueueReset**(QueueHandle_t xQueue);		Empty a queue
	BaseType_t **xQueueIsQueueEmptyFromISR**(const QueueHandle_t pxQueue);	⚡	Queries a queue if it is empty
	BaseType_t **xQueueIsQueueFullFromISR**(const QueueHandle_t pxQueue);	⚡	Queries a queue if it is full
	UBaseType_t **uxQueueMessagesWaitingFromISR**(const QueueHandle_t xQueue);	⚡	Get the number of held items in a queue
	BaseType_t **xQueueSendToBackFromISR**(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Write an item to the back of a queue
	BaseType_t **xQueueSendToFrontFromISR** (QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Write an item to the front of a queue
	BaseType_t **xQueueOverwriteFromISR** (QueueHandle_t xQueue, const void *pvItemToQueue, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Write an item in a queue even if the queue is full

	BaseType_t **xQueuePeek**(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);
	BaseType_t **xQueuePeekFromISR**(QueueHandle_t xQueue, void *pvBuffer);
		⚡	Read an item from a queue without removing it

	BaseType_t **xQueueReceive**(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);
	BaseType_t **xQueueReceiveFromISR**(QueueHandle_t xQueue, void *pvBuffer,
	BaseType_t *pxTaskWoken);
		⚡	Read an item from a queue

	UBaseType_t **uxQueueSpacesAvailable**(const QueueHandle_t xQueue);		Number of free spaces in a queue

PURPLE: CREATE functions **RED:** DELETE functions **BLUE:** Can be called within ISR's

QUEUE SET	QueueSetHandle_t **xQueueCreateSet**(const UBaseType_t uxEventQueueLength);		Create a queue set

	QueueSetMemberHandle_t **xQueueSelectFromSet**(QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait);
	QueueSetMemberHandle_t **xQueueSelectFromSetFromISR**(QueueSetHandle_t xQueueSet);
		⚡	Select a member of a queue set (queue or semaphore)

	BaseType_t **xQueueAddToSet**(QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet);		Add a queue or semaphore to a queue set
	BaseType_t **xQueueRemoveFromSet**(QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet);		Remove a queue or semaphore from a queue set
EVENT GROUPS	EventGroupHandle_t **xEventGroupCreate**(void);		Create a new event group
	void **vEventGroupDelete**(EventGroupHandle_t xEventGroup);		Delete an event group
	EventBits_t **xEventGroupGetBitsFromISR**(EventGroupHandle_t xEventGroup);	⚡	Get the bits from an event group
	EventBits_t **xEventGroupSetBitsFromISR**(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Set bits within event group
	EventBits_t **xEventGroupClearBitsFromISR**(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear);	⚡	Clear bits within event group
	EventBits_t **xEventGroupWaitBits**(const EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait);		Read bits within event group (optionally wait for a combination of set bits within same event group)
	EventBits_t **xEventGroupSync**(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait);		Set bits within event group and wait for a combination of set bits within same event group
TASK NOTIFICATION	BaseType_t **xTaskNotifyFromISR**(TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Notify a task with a 32bit value
	BaseType_t **xTaskNotifyAndQuery**(TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, uint32_t *pulPreviousNotifyValue);		Notify a task with a 32bit value and query the previous notification value
	BaseType_t **xTaskNotifyGive**(TaskHandle_t xTaskToNotify);		Notify a task and increment its notification value
	void **vTaskNotifyGiveFromISR**(TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken);	⚡	Notify a task and increment its notification value
	uint32_t **ulTaskNotifyTake**(BaseType_t xClearCountOnExit, TickType_t xTicksToWait);		Wait for a notification value (decrement or clear on exit)
	BaseType_t **xTaskNotifyWait**(uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait);		Wait for a notification value

TIMER	BaseType_t **xTimerChangePeriod**(TimerHandle_t xTimer, TickType_t xNewPeriod, TickType_t xTicksToWait);
	BaseType_t **xTimerChangePeriodFromISR**(TimerHandle_t xTimer, TickType_t xNewPeriod, BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Change the period of a timer

	TimerHandle_t **xTimerCreate**(const char *pcTimerName, const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload, void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction);		Create and initializes a new instance of a software timer
	BaseType_t **xTimerDelete**(TimerHandle_t xTimer, TickType_t xTicksToWait);		Delete a timer
	TaskHandle_t **xTimerGetTimerDaemonTaskHandle**(void);		Get the handle of the timer daemon task
	void ***pvTimerGetTimerID**(TimerHandle_t xTimer);		Get the ID of the timer
	const char * **pcTimerGetTimerName**(TimerHandle_t xTimer);		Get the Name of the timer
	BaseType_t **xTimerIsTimerActive**(TimerHandle_t xTimer);		Queries if the timer is running

	BaseType_t **xTimerPendFunctionCall**(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait);
	BaseType_t **xTimerPendFunctionCallFromISR**(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Defer the execution of a function to the RTOS daemon task

	BaseType_t **xTimerReset**(TimerHandle_t xTimer, TickType_t xTicksToWait);
	BaseType_t **xTimerResetFromISR**(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Reset a timer and start running

	void **vTimerSetTimerID**(TimerHandle_t xTimer, void *pvNewID);		Set the ID of a timer

	BaseType_t **xTimerStart**(TimerHandle_t xTimer, TickType_t xTicksToWait);
	BaseType_t **xTimerStartFromISR**(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Start a timer running

	BaseType_t **xTimerStop**(TimerHandle_t xTimer, TickType_t xTicksToWait);
	BaseType_t **xTimerStopFromISR**(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);
		⚡	Stop a timer running

MISCELLANEOUS	void **vTaskStartScheduler**(void);		Start the scheduler running
	BaseType_t **xTaskGetSchedulerState**(void);		Get the actual state of the scheduler
	void **vTaskList**(char *pcWriteBuffer);		Get a short overview of all tasks
	void **vTaskGetRunTimeStats**(char *pcWriteBuffer);		Get run time statistics
	UBaseType_t **uxTaskGetSystemState**(TaskStatus_t * const pxTaskStatusArray, const UBaseType_t uxArraySize, unsigned long * const pulTotalRunTime);		Get information's of all existing tasks
	TaskHandle_t **xTaskGetIdleTaskHandle**(void);		Get the handle associated to the idle task
	TickType_t **xTaskGetTickCountFromISR**(void);	⚡	Get the current tick counter value

PURPLE: CREATE functions **RED:** DELETE functions **BLUE:** Can be called within ISR's