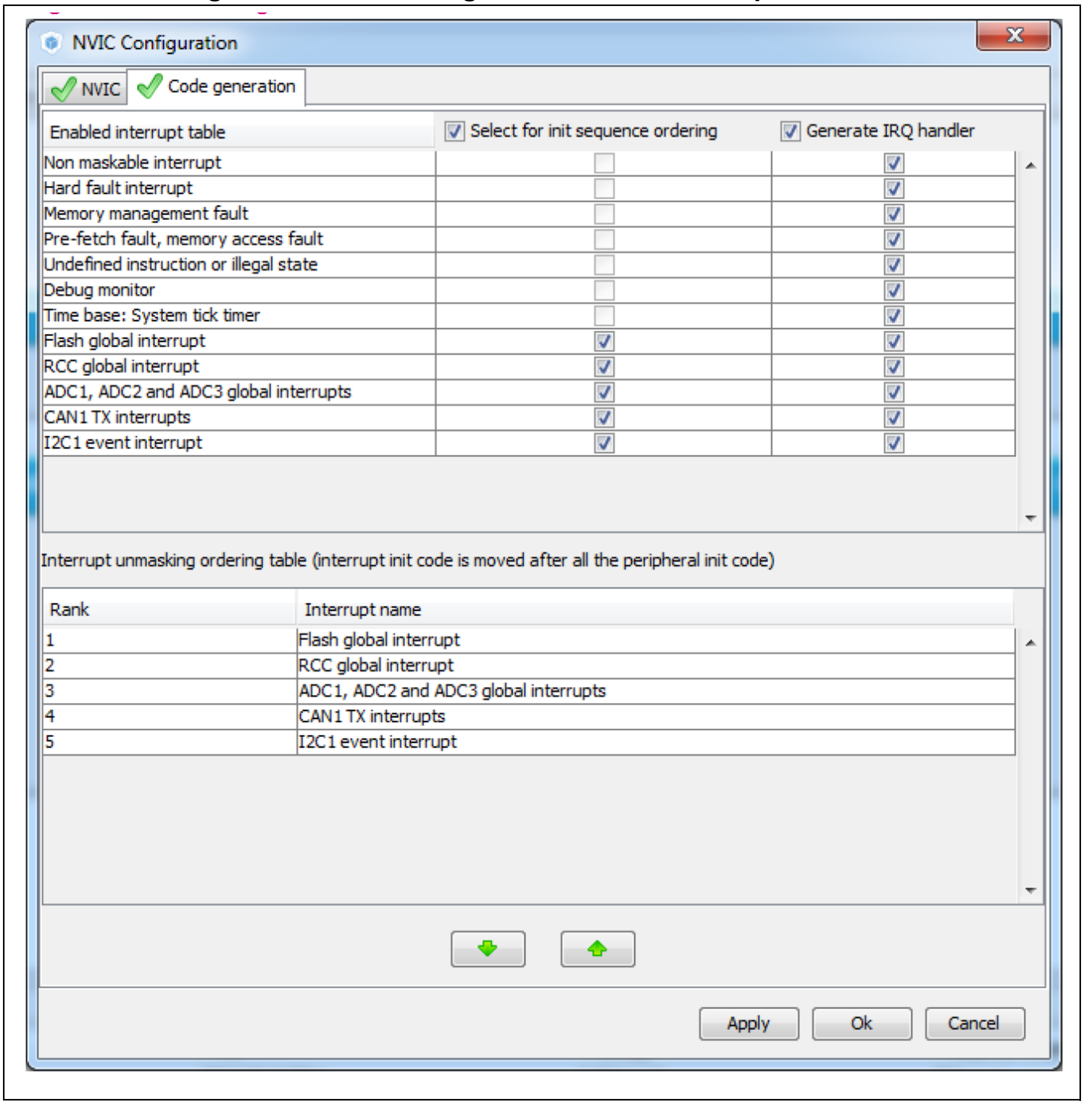
Figure 80. NVIC Code generation – All interrupts enabled



- **Default initialization sequence of interrupts**

  By default, the interrupts are enabled as part of the peripheral MSP initialization function, after the configuration of the GPIOs and the enabling of the peripheral clock.

  This is shown in the CAN example below, where HAL_NVIC_SetPriority and HAL_NVIC_EnableIRQ functions are called within stm32xxx_hal_msp.c file inside the peripheral msp_init function.

  Interrupt enabling code is shown in green.

```
void HAL_CAN_MspInit(CAN_HandleTypeDef* hcan)
{
GPIO_InitTypeDef GPIO_InitStruct;
if(hcan->Instance==CAN1)
{
 /* Peripheral clock enable */
 __CAN1_CLK_ENABLE();
 /**CAN1 GPIO Configuration
```

```
  PD0     ------> CAN1_RX
  PD1     ------> CAN1_TX
  */
  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF9_CAN1;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

/* Peripheral interrupt init */
  HAL_NVIC_SetPriority(CAN1_TX_IRQn, 2, 2);
  HAL_NVIC_EnableIRQ(CAN1_TX_IRQn);
  }
 }
```

For **EXTI GPIOs** only, interrupts are enabled within the MX_GPIO_Init function:

```
/*Configure GPIO pin : MEMS_INT2_Pin */
 GPIO_InitStruct.Pin = MEMS_INT2_Pin;
 GPIO_InitStruct.Mode = GPIO_MODE_EVT_RISING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(MEMS_INT2_GPIO_Port, &GPIO_InitStruct);

 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

For some peripherals, the application still needs to call another function to actually activate the interruptions. Taking the timer peripheral as an example, the function HAL_TIM_IC_Start_IT needs to be called to start the Timer input capture (IC) measurement in interrupt mode.

- **Configuration of interrupts initialization sequence**

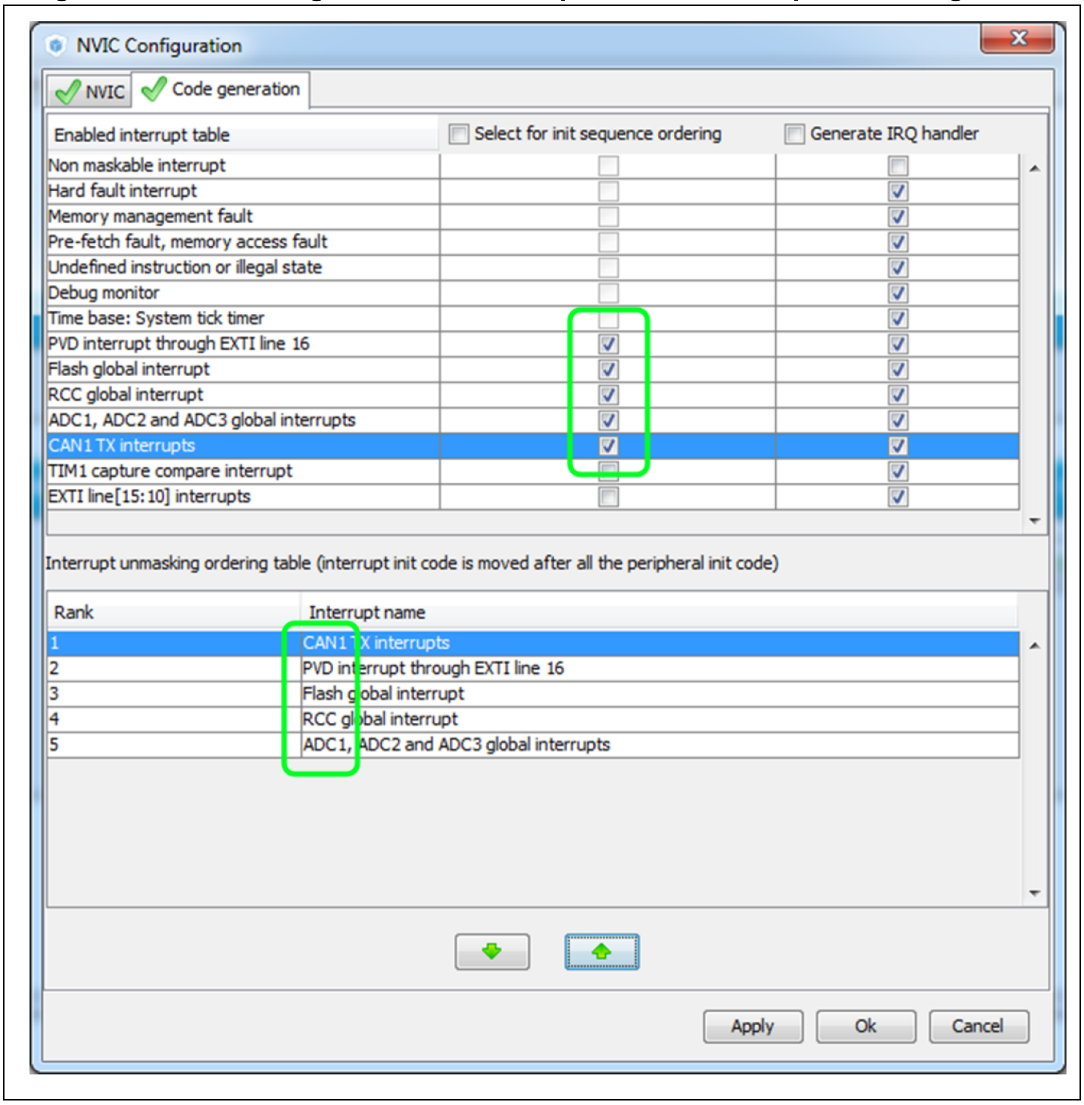
  Checking **Select for Init sequence ordering** for a set of peripherals moves the HAL_NVIC function calls for each peripheral to a same dedicated function, named **MX_NVIC_Init**, defined in the main.c. Moreover, the HAL_NVIC functions for each peripheral are called in the order specified in the **Code generation** view bottom part (see *Figure 81*).

  As an example, the configuration shown in *Figure 81* generates the following code:

```
/** NVIC Configuration
*/
void MX_NVIC_Init(void)
{
 /* CAN1_TX_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(CAN1_TX_IRQn, 2, 2);
 HAL_NVIC_EnableIRQ(CAN1_TX_IRQn);
 /* PVD_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(PVD_IRQn, 0, 0);
```

```
 HAL_NVIC_EnableIRQ(PVD_IRQn);
 /* FLASH_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(FLASH_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(CAN1_IRQn);
 /* RCC_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(RCC_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(CAN1_IRQn);
 /* ADC_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(ADC_IRQn);
}
```

Figure 81. NVIC Code generation – Interrupt initialization sequence configuration



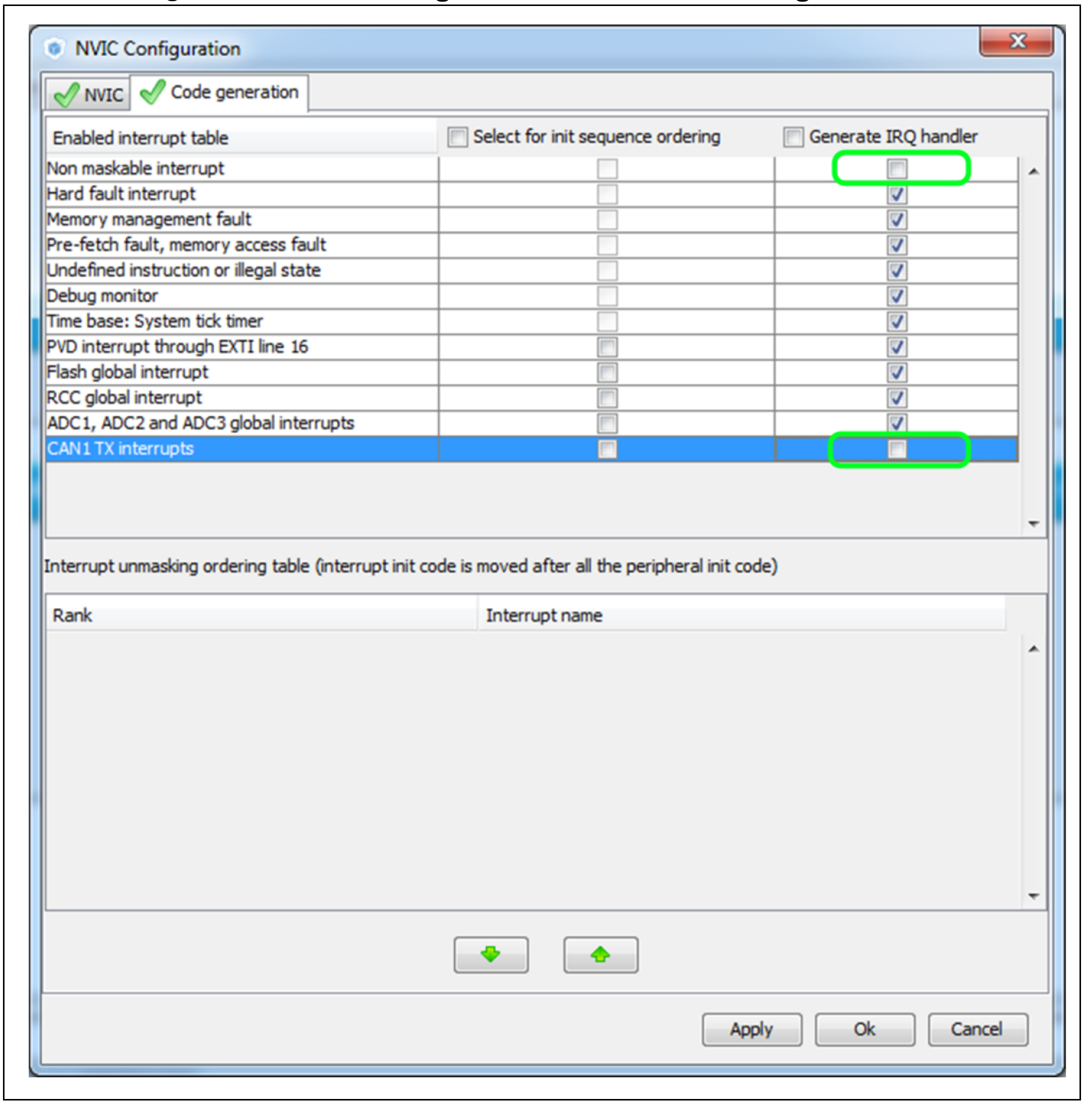
- **Interrupts handler code generation**

By default, STM32CubeMX generates interrupt handlers within the stm32xxx_it.c file. As an example:

```
void NMI_Handler(void)
{
 HAL_RCC_NMI_IRQHandler();
}
void CAN1_TX_IRQHandler(void)
{
 HAL_CAN_IRQHandler(&hcan1);
}
```

The column **Generate IRQ Handler** allows controlling whether the interrupt handler function call shall be generated or not. Unselecting CAN1_TX and NMI interrupts from the **Generate IRQ Handler** column as shown in *Figure 81* removes the code mentioned earlier from the stm32xxx_it.c file.

Figure 82. NVIC Code generation – IRQ Handler generation



4.12.6 FreeRTOS middleware configuration view

Through STM32CubeMX FreeRTOS configuration window, the user can configure all the resources required for a real-time OS application and reserve the corresponding heap. FreeRTOS elements are defined and created in the generated code using CMSIS-RTOS API functions. Follow the sequence below:

1. In the **Configuration** tab, enable FreeRTOS from the tree view.
2. Click FreeRTOS in the **Configuration** pane to open the FreeRTOS configuration window (see *Figure 83*).

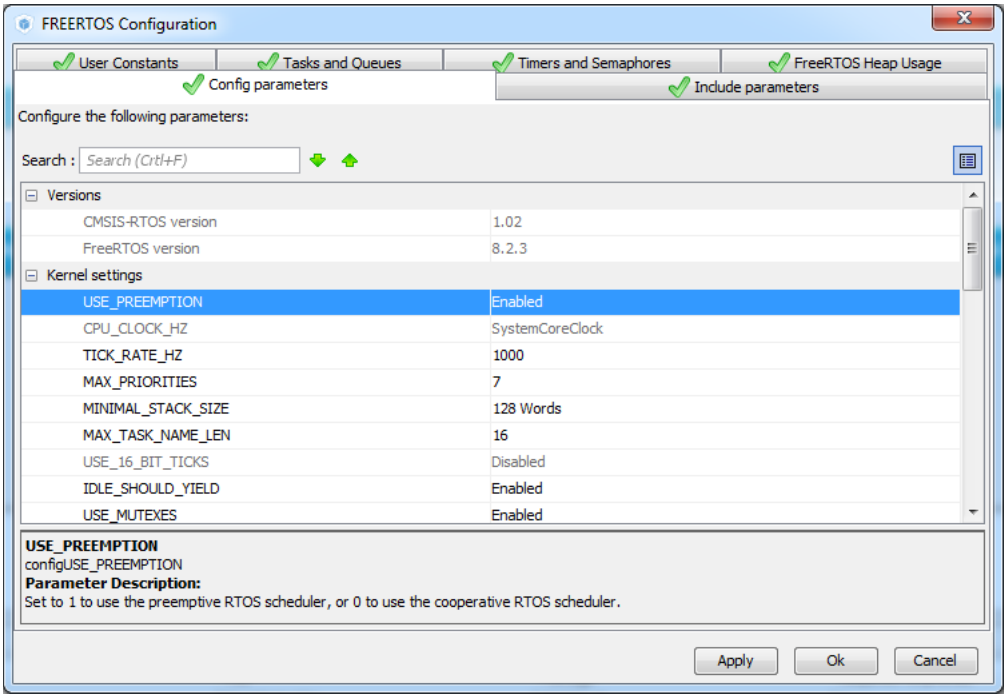
   All tabs but the **User Constants** tab allow configuring FreeRTOS native configuration parameters and objects, such as tasks, timers, queues, and semaphores.

   The **Config parameters** values allow configuring Kernel and Software settings.

   The **Include parameters** tab allows selecting only the API functions required by the application and thus optimizing the code size.

   Both Config and Include parameters will be part of the **FreeRTOSConfig.h** file.

Figure 83. FreeRTOS configuration view

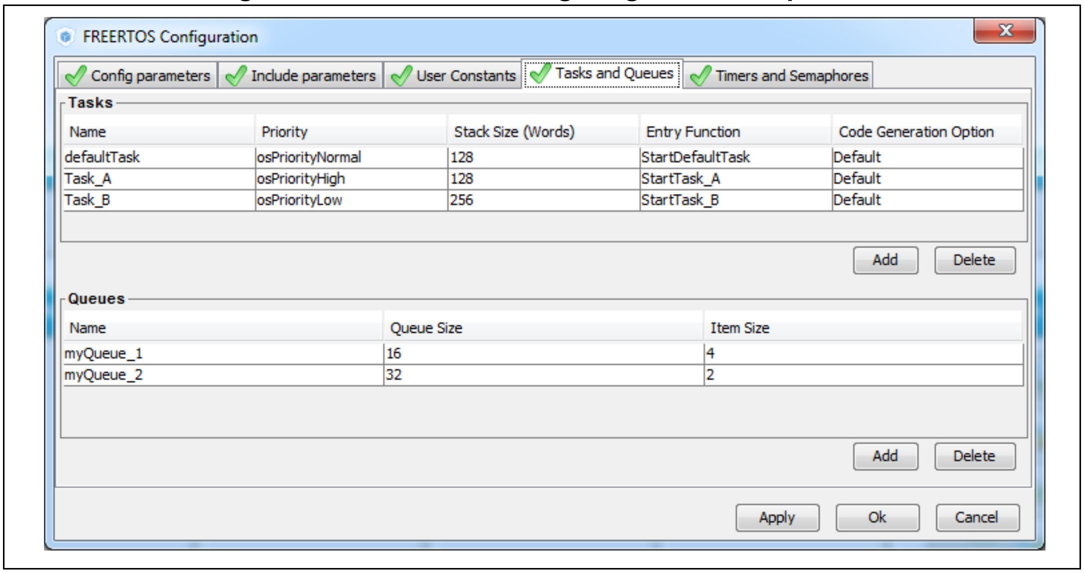


Tasks and Queues Tab

As any RTOS, FreeRTOS allows structuring a real-time application into a set of independent tasks, with only one task being executed at a given time. Queues are meant for inter-task communications: they allow to exchange messages between tasks or between interrupts and tasks.

In STM32CubeMX, the **FreeRTOS Tasks and Queues** tab allows creating and configuring such tasks and queues (see *Figure 84*). The corresponding initialization code will be generated within main.c or freeRTOS.c if the project option to generate per IP is selected (see *Figure 40: Project Settings Code Generator*).

Figure 84. FreeRTOS: configuring tasks and queues



- Tasks

  Under the **Tasks** section, click the Add button to open the **New Task** window where task **name**, **priority**, **stack size** and **entry function** can be configured (see *Figure 85*). These settings can be updated at any time: double-clicking a task row opens again the new task window for editing.

  The entry function can be generated as weak or external:

  – When the task is generated as **weak**, the user can propose another definition than the one generated by default.

  – When the task is **extern**, it is up to the user to provide its function definition.

  By default, the function definition is generated including user sections to allow customization.
- Queues

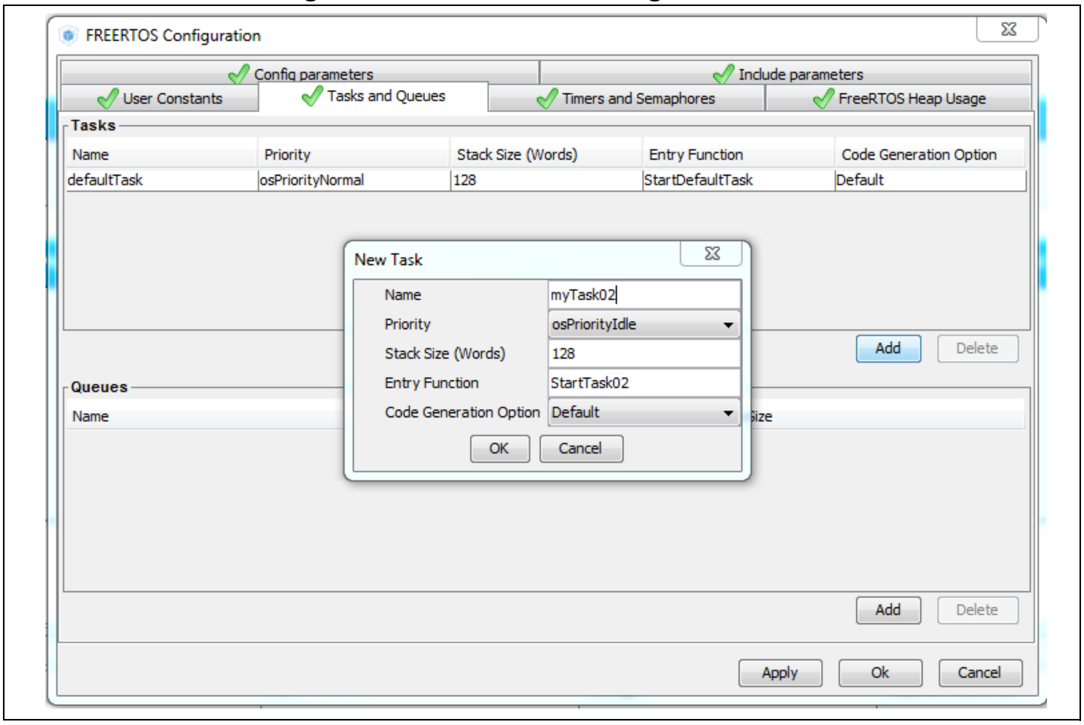
  Under the **Queues** section, click the **Add** button to open the **New Queue** window where the queue **name**, **size** and **item size** can be configured (see *Figure 85*). The queue size corresponds to the maximum number of items that the queue can hold at a time, while the item size is the size of each data item stored in the queue. The item size can be expressed either in number of bytes or as a data type:

- 1 byte for uint8_t, int8_t, char and portCHAR types
- 2 bytes for uint16_t, int16_t, short and portSHORT types
- 4 bytes for uint32_t, int32_t, int, long and float
- 8 bytes for uint64_t, int64_t and double

A default value of 4 bytes will be used when the item size can not be automatically derived from user input.

These settings can be updated at any time: double-clicking a queue row opens again the new queue window for editing.

Figure 85. FreeRTOS: creating a new task



The following code snippet shows the generated code corresponding to *Figure 84: FreeRTOS: configuring tasks and queues*.

```
/* Create the thread(s) */
  /* definition and creation of defaultTask */
  osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
  defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

  /* definition and creation of Task_A */
  osThreadDef(Task_A, StartTask_A, osPriorityHigh, 0, 128);
  Task_AHandle = osThreadCreate(osThread(Task_A), NULL);

  /* definition and creation of Task_B */
  osThreadDef(Task_B, StartTask_B, osPriorityLow, 0, 256);
```

```
Task_BHandle = osThreadCreate(osThread(Task_B), NULL);

/* Create the queue(s) */
/* definition and creation of myQueue_1 */
osMessageQDef(myQueue_1, 16, 4);
myQueue_1Handle = osMessageCreate(osMessageQ(myQueue_1), NULL);

/* definition and creation of myQueue_2 */
osMessageQDef(myQueue_2, 32, 2);
myQueue_2Handle = osMessageCreate(osMessageQ(myQueue_2), NULL);
```

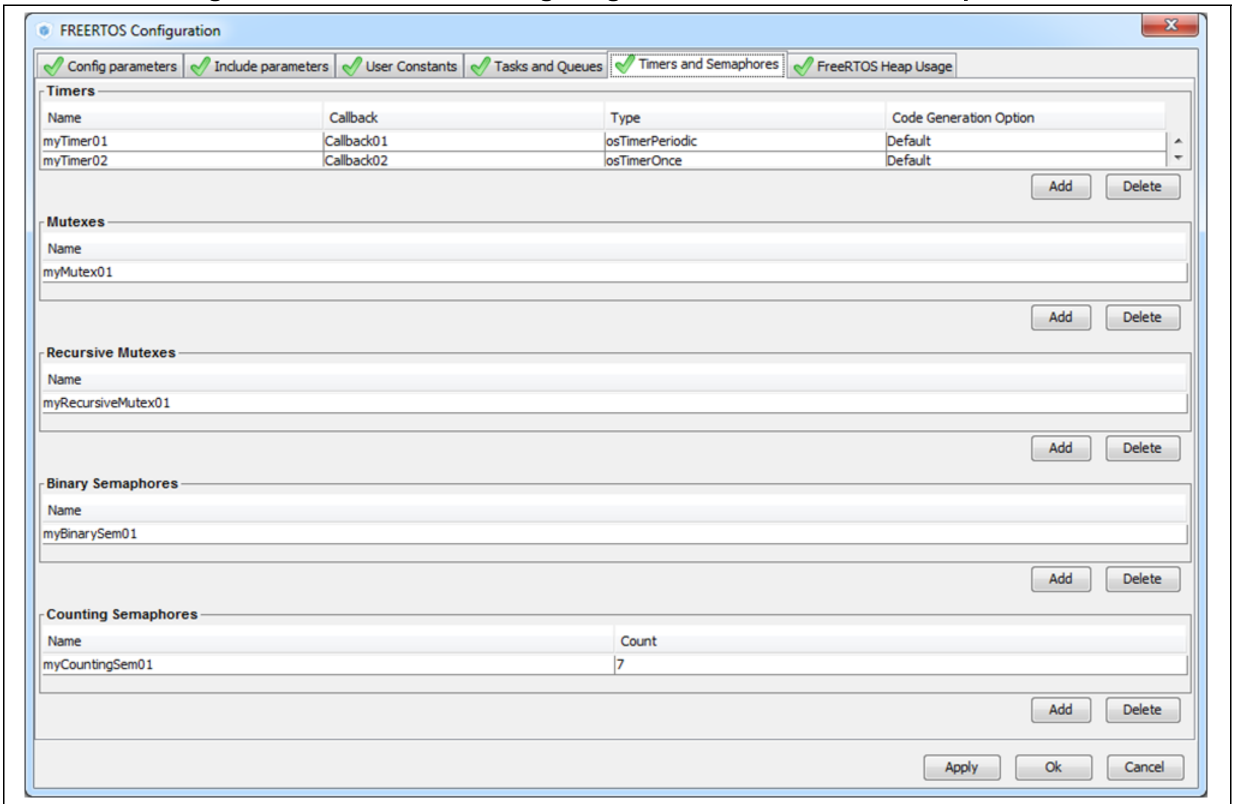
Timers, Mutexes and Semaphores

FreeRTOS timers, mutexes and semaphores can be configured via the FreeRTOS **Timers and Semaphores** tab (see *Figure 86*).

Under each object dedicated section, clicking the **Add** button to open the corresponding **New <object>** window where the object specific parameters can be specified. Object settings can be modified at any time: double clicking the relevant row opens again the **New <object>** window for edition.

Note: Expand the window if the newly created objects are not visible.

Figure 86. FreeRTOS - Configuring timers, mutexes and semaphores



- Timers

  Prior to creating timers, their usage (USE_TIMERS definition) must be enabled in the **software timer definitions section** of the **Configuration parameters** tab. In the same section, timer task priority, queue length and stack depth can be also configured.

  The timer can be created to be one-shot (run once) or auto-reload (periodic). The timer name and the corresponding callback function name must be specified. It is up to the user to fill the callback function code and to specify the timer period (time between the timer being started and its callback function being executed) when calling the CMSIS-RTOS osTimerStart function.

- Mutexes/Semaphores

  Prior to creating mutexes, recursive mutexes and counting semaphores, their usage (USE_ MUTEXES, USE_RECURSIVE_MUTEXES, USE_COUNTING_SEMAPHORES definitions) must be enabled within the **Kernel settings** section of the **Configuration parameters** tab.

  The following code snippet shows the generated code corresponding to *Figure 86: FreeRTOS - Configuring timers, mutexes and semaphores*).

```
/* Create the semaphores(s) */
/* definition and creation of myBinarySem01 */
osSemaphoreDef(myBinarySem01);
myBinarySem01Handle = osSemaphoreCreate(osSemaphore(myBinarySem01), 1);

/* definition and creation of myCountingSem01 */
osSemaphoreDef(myCountingSem01);
myCountingSem01Handle = osSemaphoreCreate(osSemaphore(myCountingSem01),
7);

  /* Create the timer(s) */
/* definition and creation of myTimer01 */
osTimerDef(myTimer01, Callback01);
myTimer01Handle = osTimerCreate(osTimer(myTimer01), osTimerPeriodic,
NULL);

/* definition and creation of myTimer02 */
osTimerDef(myTimer02, Callback02);
myTimer02Handle = osTimerCreate(osTimer(myTimer02), osTimerOnce, NULL);

/* Create the mutex(es) */
/* definition and creation of myMutex01 */
osMutexDef(myMutex01);
myMutex01Handle = osMutexCreate(osMutex(myMutex01));

/* Create the recursive mutex(es) */
/* definition and creation of myRecursiveMutex01 */
osMutexDef(myRecursiveMutex01);
```

```
  myRecursiveMutex01Handle =
osRecursiveMutexCreate(osMutex(myRecursiveMutex01));
```

FreeRTOS heap usage

The **FreeRTOS Heap usage** tab displays the heap currently used and compares it to the TOTAL_HEAP_SIZE parameter set in the **Config Parameters** tab. When the total heap used crosses the TOTAL_HEAP_SIZE maximum threshold, it is shown in red and a red cross appears on the tab (see *Figure 87*).

Figure 87. FreeRTOS Heap usage