SMART ARM-based Microcontrollers

Atmel AT13723:Getting Started with FreeRTOS on Atmel SAMV/S/E MCUs

APPLICATION NOTE

Introduction

This application note illustrates the basic functionality of the FreeRTOS™ Real Time Operating System and shows how to use it on Atmel® | SMART SAM V/S/E microcontrollers.

This application note covers:

- What is a Real-Time application and a real time operating system?
- How to create and configure a FreeRTOS project
- How to make use of FreeRTOS basic functionality in an embedded project

The description is based on FreeRTOS port available for SAM V71 Xplained Ultra board in FreeRTOS.org™ website. All the processes illustrated in this document is explained with an example from SAMV71-XULT Atmel Studio Software Package 1.3.

Table of Contents

1. Introduction

1.1. What is a Real-time Application?

The main difference between a standard application and a real-time application is the time constraint related to actions to perform. In a real-time application the time by which tasks will execute can be predicted deterministically on the basis of knowledge about the system's hardware and software. Typically, applications of this type include a mix of both hard and soft real-time requirements.

Soft real-time requirements: are those that state a time deadline but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly may make a system seem annoyingly unresponsive without actually making it unusable.

Hard real-time requirements: are those that state a time deadline and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag would be useless if it responded to crash sensor inputs too slowly.

To adhere to these time requirements, the usage of a real time operating system (RTOS) is often required.

1.2. Real-time Operating System and Multitasking

The most basic feature, common to all operating system is the support for multitasking. Additional features that support the essential components such as networking, peripheral interfacing, user interface and printing can be added

An embedded system may not require all of these components. The types of operating systems used in real time embedded system often have only the fundamental function of support for multitasking. These operating systems can vary in size from 300bytes to 10KB. So they are small enough to fit inside internal microcontroller flash memory.

Embedded systems usually have access to only one processor, which serve many input and output paths. Real time operating system must divide time between various activities such that all the deadlines (requirements) are met.

A real time operating system always includes:

- Support of multiple task running concurrently
- A scheduler to determine which task should run
- Ability for the scheduler to pre-empt a running task
- Support for inter-task communication

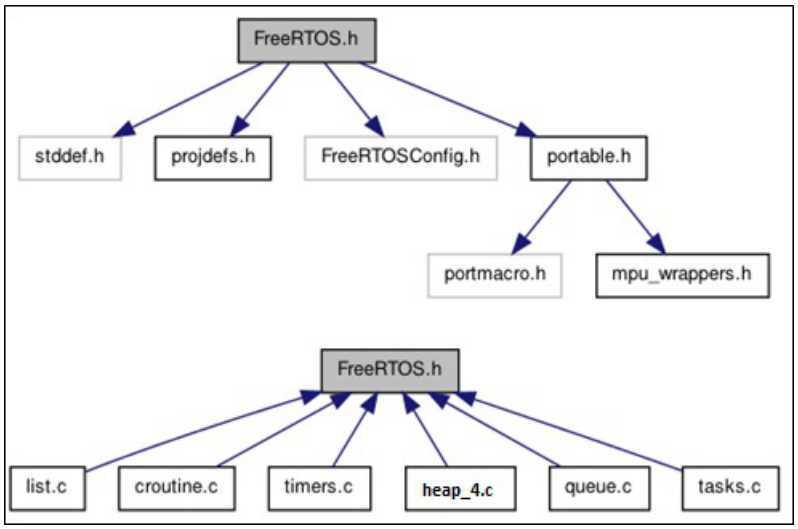
1.3. FreeRTOS Introduction

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which Cortex®-M7 microcontroller applications can be built to meet their hard real-time requirements. It allows Cortex-M7 microcontroller applications to be organized as a collection of independent tasks to be executed. As Cortex-M7 microcontrollers from Atmel have only one core, only one task can be executed at a time. The kernel decides which task should be executing by examining the priority assigned to each by the application designer. In the simplest case, the application designer could assign higher priorities to tasks that implement hard real-time requirements and lower priorities to tasks that implement soft real-time requirements. This would ensure that hard real-time tasks are always executed ahead of soft real-time one.

1.3.1. The FreeRTOS Kernel

FreeRTOS kernel is target independent. This kernel is ported for many microcontrollers and distributed through FreeRTOS website itself as well as from the microcontroller vendors separately. For SAM V7 Xplained Ultra kit, this FreeRTOS kernel is ported and also a demo is available in FreeRTOS website. This kernel files can be added manually in the software package so that this RTOS based application can be developed easily.

Figure 1-1 FreeRTOS Module Organization



The Cortex-M7 port include all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Software timers
- Queues
- Binary semaphores
- Counting semaphores
- Recursive semaphores
- Mutexes
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace hook macros

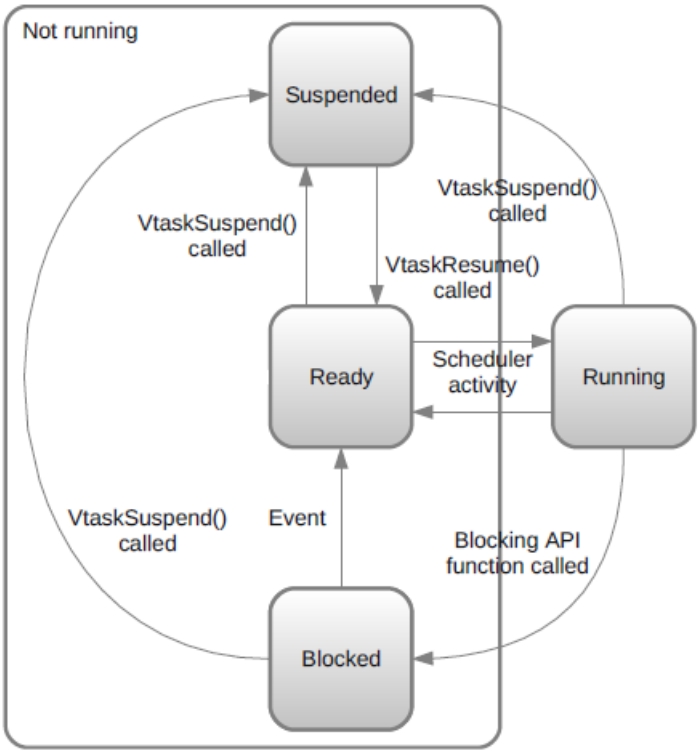
FreeRTOS can be configured to exclude unused functionality from compiling and thereby reducing its memory footprint.

Note: The FreeRTOS kernel is released under GPL with exception, allowing user applications to stay closed source. The BSP part is a mix of GPL with exception license and code provided by the different hardware manufacturers

1.3.2. FreeRTOS Taks Management Mechanism

FreeRTOS allows to handle multiple concurrent tasks, but only one task can be run at a time (single core processor). Thus the system requires a scheduler to time slice the execution of concurrent tasks. The scheduler is the core of the FreeRTOS kernel; it selects the task to be executed according to priority and state of the task. The various task states are illustrated in following figure.

Figure 1-2 Various Tasks of the FreeRTOS Kernel

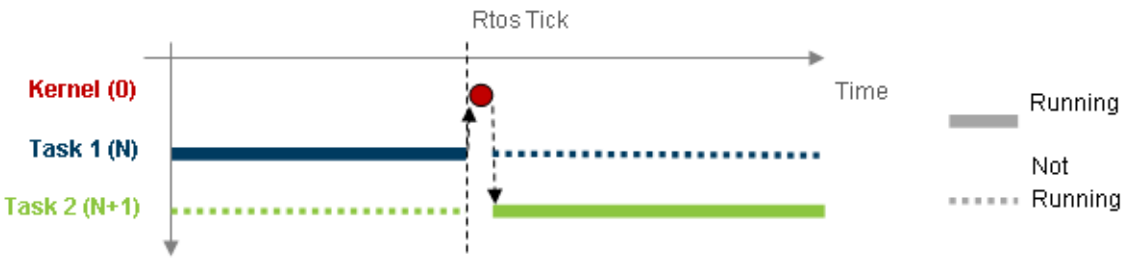


At application level there are two possible states for a task: **Running** and **Not Running**. At scheduler level, **Not Running** state is divided in to three categories:

1. **Suspend**: Task has been suspended (deactivated) by the application
2. **Blocked**: Task is blocked and waiting for synchronization event
3. **Ready**: Ready to execute, but a task with higher priority is running

Task scheduling aims to decide which task in **Ready** state has to be run at a given time. FreeRTOS achieves this purpose with priorities given to tasks while they are created. Priority of a task is the only element the scheduler takes into account to decide which task has to be switched in. Every clock tick makes the scheduler to decide which task has to be woken up.

Figure 1-3 RTOS Tick and a Task State

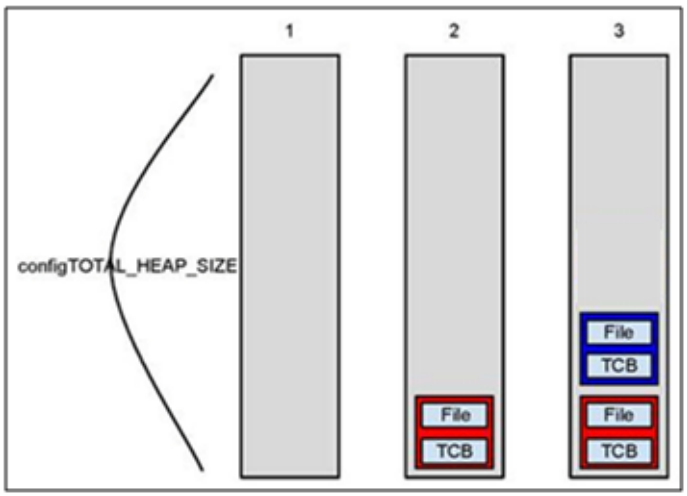


1.3.3. FreeRTOS Memory Management

FreeRTOS allows unlimited number of tasks to be executed as long as hardware and memory can handle them. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks.

The memory allocation of tasks in RAM is illustrated in the following figure.

Figure 1-4 The Memory Allocation of Tasks in RAM



The RTOS kernel allocates RAM each time a task or a kernel object is created. The section allocated to a task or an object is called a stack. The size of this stack is configurable at task creation. The stack contains the *Task File* and the *Task Control Board* (TCB) that allows the kernel to handle the task. All stacks are stored in a section called HEAP. The heap management is done according the `Heap_x.c` file included with the kernel the selection of `Heap_x.c` file should be done according to application requirement.

- **Heap_1.c**: This is the simplest implementation of all. It does not permit memory to be freed once it has been allocated.
- **Heap_2.c**: This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does not however combine adjacent free blocks into a single large block.
- **Heap_3.c**: This implements a simple wrapper for the standard C library `malloc()` and `free()` functions that will, in most cases, be supplied with your chosen compiler. The wrapper simply makes the `malloc()` and `free()` functions thread safe.
- **Heap_4.c**: This scheme uses a first fit algorithm and, unlike scheme 2, does combine adjacent free memory blocks into a single large block (it does include a coalescence algorithm).
- **Heap_5.c**: This scheme uses the same first fit and memory coalescence algorithms as heap_4, and allows the heap to span multiple non adjacent (non-contiguous) memory regions.

In all cases except `Heap_3.c`, the total amount of available heap space is set by *configTOTAL_HEAP_SIZE* defined in `FreeRTOSConfig.h`. In case of scheme 3, the heap size is configured in linker script.

2. FreeRTOS Kernel Inclusion and Configuration

This section describes the inclusion and configuration of FreeRTOS kernel in an Atmel studio 6.2 project. The following aspects are covered:

- Opening example project from SAMV71-XULT Atmel Studio Software Package
- Manually adding the FreeRTOS kernel in an existing project
- Configure FreeRTOS kernel according to product specification
- Optimize kernel size according to application requirements

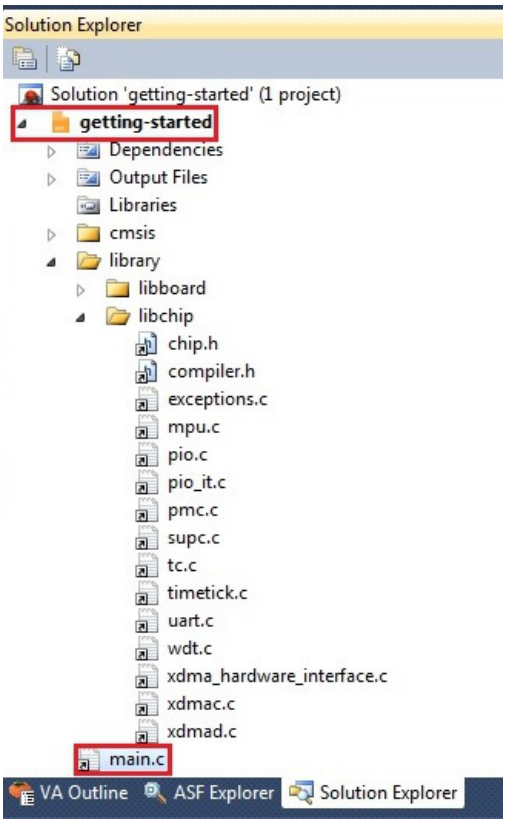
The FreeRTOS Website (Version 8.2.1 and above) contains a Demo project for SAM V71 Xplained Ultra board in Atmel Studio, IAR and Keil platforms. This demo is useful to get started with FreeRTOS and SAM V71 device. The above mentioned steps will be helpful in adding FreeRTOS library in software package. This process explains the steps involved in making existing example projects into a RTOS based application.

2.1. SAMV71-XULT Atmel Studio Software Package

Peripheral drivers, middleware libraries, and example projects for SAMV71-XULT board are available in the software package. In Atmel Studio website, software packages for SAM V71 Xplained Ultra board are available for IAR, Keil, GCC, and Atmel Studio platforms.

All example projects uses common drivers available in the libraries folder. All driver files are added in the project as links. Opening an example project from this software package will display the folder and file structure. The following figure displays the files and folder structure of the getting started example project.

Figure 2-1 Example Project Folder Structure



FreeRTOS source files, include files, and configuration files must be manually taken from the FreeRTOS port and added into this software package. Then these FreeRTOS files can be manually included (like other library files already included) in any project, if needed.

For example, getting started example project from Atmel Studio software package is used in this application note to demonstrate the addition of FreeRTOS files. Software package has example projects for many peripherals. Getting started example is available in following path in software package.

```
$Install_Directory$\samv71_softpack_1.3_for_astudio_6_2\studio\Atmel
\samv71_Xplained_Ultra\examples\getting-started\build\studio
```

2.2. Add the FreeRTOS Library to Software Package

The FreeRTOS kernel and port for Cortex-M7 is available from FreeRTOS Version 8.2.1. Downloaded source files from FreeRTOS website contains port and examples for many devices. Support for Cortex-M7 is added from version 8.2.1 and basic example is available for SAM V71 Xplained Ultra board. So we can identify the source files required for SAM V71 device from FreeRTOS library. Add a folder for FreeRTOS in software package libraries folder and add the required FreeRTOS files in that folder in the same folder structure available in FreeRTOS source. The following figure shows the folder structure that has to be created in software package libraries folder.

Figure 2-2 FreeRTOS Library Folder in Software Package

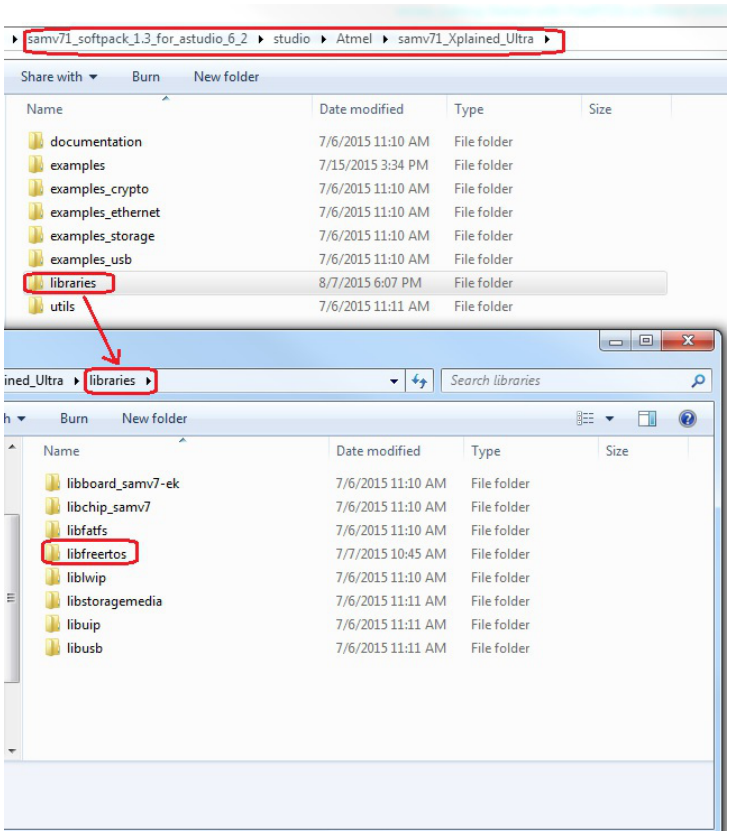
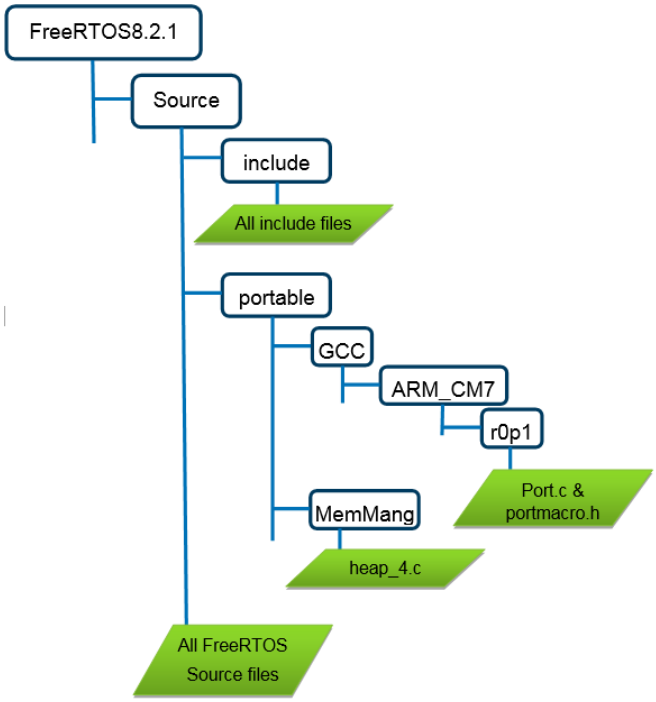


Figure 2-3 FreeRTOS Folder and File Structure



2.3. Include the Added FreeRTOS files in an Example Project

In an Example project, FreeRTOS files can be included as link. Add the same FreeRTOS folder structure in the Atmel Studio project first. In each folder add the recently added freertos files available from software package. While adding the files, option **Add as Link** is used so that libraries files are only available in a single place and no redundant copies are created inside the project folder.

Figure 2-4 Including a File in a Project

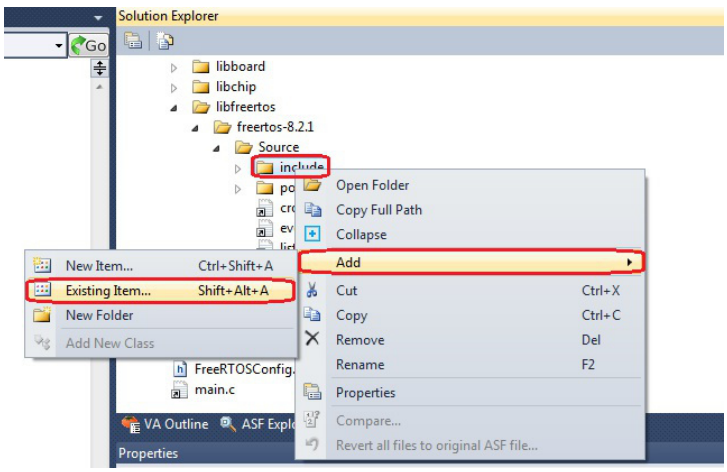
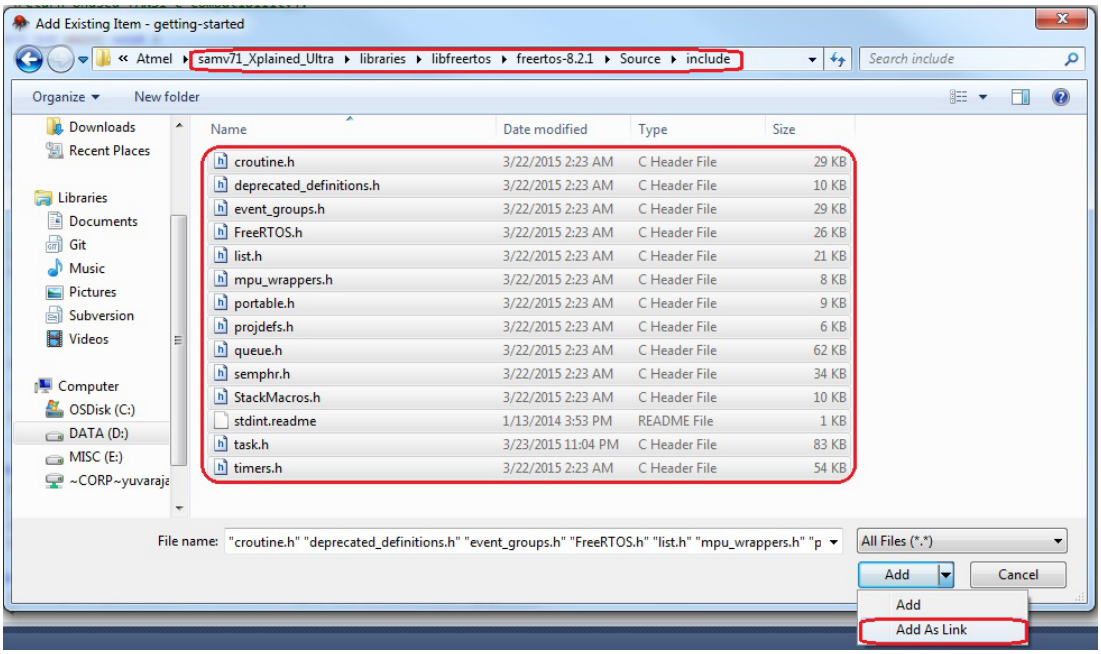
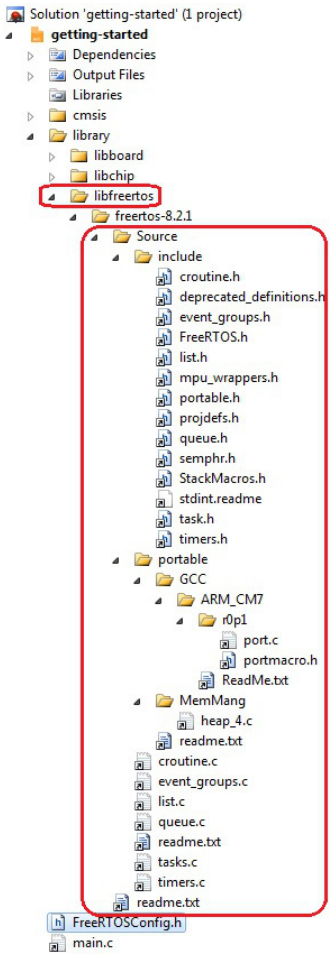


Figure 2-5 Adding Files as Link in Atmel Studio Project



After including all FreeRTOS files, compiler directories should be updated with relative path for each FreeRTOS folder to avoid errors while compiling. Add compiler flag *-mfpu=fpv5-sp-d16 -mfloat-abi=softfp* in compiler **Miscellaneous** option in **Project Settings**. This compiler flag is the same flag used in FreeRTOS Demo project for Atmel Studio.

Figure 2-6 FreeRTOS Folder and File Structure



The kernel configuration is performed through a dedicated header file `FreeRTOSConfig.h`. This file is application dependent as each application might have different kernel requirement. This file can be added along with other configuration files in project or can be placed along with main file.

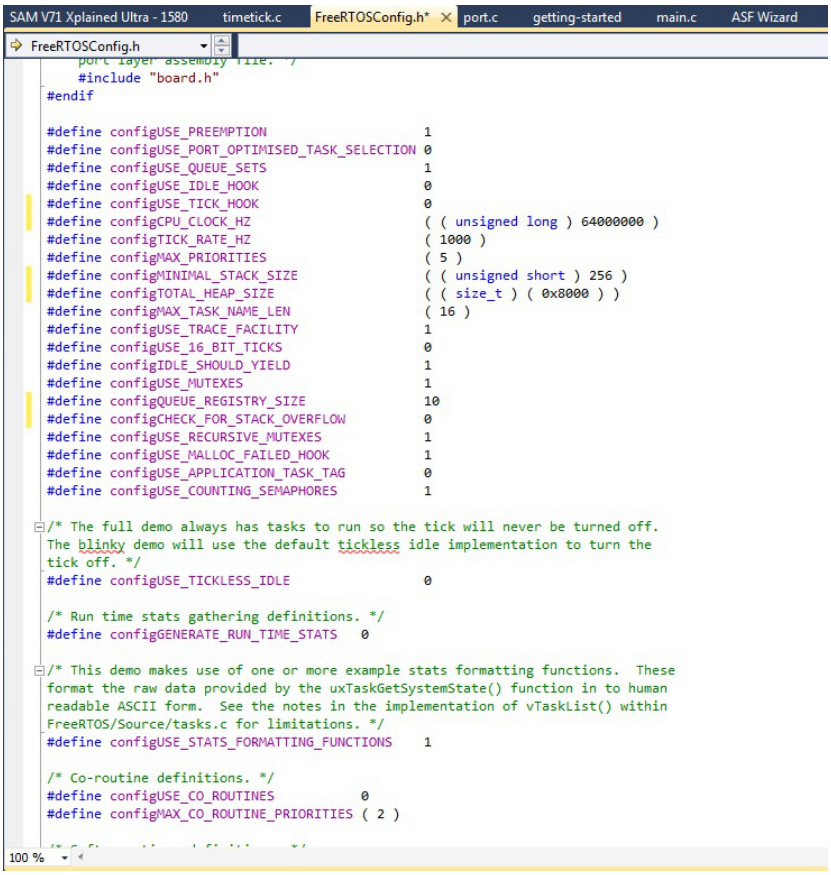
This file is available in ARM_CortexM7 Example FreeRTOS folder in following path `FreeRTOSV8.2.1\FreeRTOS\Demo\CORTEX_M7_SAMV71_Xplained_IAR_Keil`

Note: FreeRTOS source code is licensed by the modified GNU General Public License (GPL). A license agreement is required to use the kernel in an commercial project. Information about this license can be read in the license agreement window that appears when adding the kernel.

2.4. Configuring the Kernel according to Application Requirement

The kernel configuration is performed through a dedicated header file `FreeRTOSConfig.h` available in the project. The kernel configuration is achieved by modifying some predefine *config* and *INCLUDE* definitions. By default these definition are already configured.

Figure 2-7 FreeRTOSConfig.h file



Taking time to adapt the kernel to application needs allows reducing the footprint of the kernel in memory. Following table lists the FreeRTOS kernel configuration and customization definitions that can be found in the `FreeRTOSConfig.h`. FreeRTOS website has dedicated webpage for explaining the FreeRTOSConfig.h parameters. This webpage gives information on all configuration parameters. Based on the configurations defined, sometime dummy or valid definition needs to be added in main for configurations like `vApplicationTickHook()`, `vApplicationMallocFailedHook()` etc. Demo application for Atmel Studio in FreeRTOS website is also a useful reference for this

Table 2-1 FreeRTOS Configuration and Customization Definitions

Config definition	Description
configUSE_PREEMPTION	Set to 1 to use the preemptive RTOS scheduler, or 0 to use the cooperative RTOS scheduler
configUSE_IDLE_HOOK	Enable/disable IDLE Hook (callback when system has no active task)
configUSE_TICK_HOOK	Enable/disable TICK Hook (callback on every tick)
configCPU_CLOCK_HZ	Defines CPU clock for tick generation
configTICK_RATE_HZ	Defines Tick Frequency in Hertz
configMAX_PRIORITIES	Defines the number priority level that kernel need to manage
configMINIMAL_STACK_SIZE	Defines the minimal stack size allocated to a task

Config definition	Description
configTOTAL_HEAP_SIZE	Defines the size of the system heap
configMAX_TASK_NAME_LEN	Defines the Maximum Task name length (used for debug)
configUSE_TRACE_FACILITY	Build/omit Trace facility (used for debug)
configUSE_16_BIT_TICKS	1: portTickType = uint_16; 0: portTickType = uint_32 Improve performance of the system, but Impact the maximum time a task can be delayed
configIDLE_SHOULD_YIELD	The users application creates tasks that run at the idle priority
configUSE_MUTEXES	Build/omit Mutex support functions
configQUEUE_REGISTRY_SIZE	Defines the maximum number of queues and semaphores that can be registered
configCHECK_FOR_STACK_OVERFLOW	Enables stack over flow detection
configUSE_RECURSIVE_MUTEXES	Build/omit Recursive Mutex support functions
configUSE_MALLOC_FAILED_HOOK	Build/omit Malloc failed support functions
configUSE_APPLICATION_TASK_TAG	Build/omit Task tag functions
configUSE_COUNTING_SEMAPHORES	Build/omit counting semaphore support functions
configUSE_CO_ROUTINES	Build/omit co-routines support functions
configMAX_CO_ROUTINE_PRIORITIES	Defines the maximum level of priority for coroutines
configUSE_TIMERS	Build/omit timers support functions
configTIMER_TASK_PRIORITY	Defines timer task priority level
configTIMER_QUEUE_LENGTH	Sets the length of the software timer command queue
configTIMER_TASK_STACK_DEPTH	Sets the stack depth allocated to the software timer service/ daemon task

2.4.1. System and Tick Frequency

An important point to take in account when using an RTOS is the system frequency and more particularly the kernel tick frequency (Time base information of the RTOS). The kernel tick frequency is defined in the `FreeRTOSConfig.h` and is based by default on the MCU frequency. The tick frequency can be set according to following definitions: MCU Frequency is 64MHz and tick rate is 1kHz for example shown here.

```
          #define configCPU_CLOCK_HZ                ((unsigned
long)64000000)
          #define configTICK_RATE_HZ                ((portTickType)1000)
```

3. Tasks Creation and Scheduling

This section describes the basic tasks creation, scheduling, and handling processes:

- Task structure
- Task creation
- Task scheduling
- Priority setting

All steps are illustrated by graphic trace kind of application to visualize the impact of the different kernel function calls and settings.

3.1. Task Structure

A task is implemented by a function that should never return. They are typically implemented as a continuous loop such as in the `vATaskFunction` shown in the following example:

```
void vATaskFunction( void *pvParameters )
{
       for( ;; )
    {
        /* Task application code here.*/
    }
}
```

Since there is no value to be returned, the task should be of a void type. A specific structure `pvParameters` can be used to pass information of any type into the task:

```
typedef struct {
    const char Parameter1;
    const uint32_t Parameter2;
    /*...*/
} pvParameters;
```

A task must be created (Memory allocation + add to Scheduling list). During creation, a handler ID is assigned to each task. This ID is used as parameter for all kernel task management function.

```
xTaskHandle task_handle_ID;
```

3.2. Task Creation and Deletion

The task creation and deletion are performed by kernel function `xTaskCreate()` and `xTaskDelete()`

3.2.1. xTaskCreate Function

The `xTaskCreate` function creates a task by allocating RAM to it (creation of the task stack). It's parameters allows to set the name, stack depth, and priority of the task, and also to retrieve task identifier and pointer to RAM function where the task code is implemented. After the creation a task is ready to be executed. The `xTaskCreate` function call should be done prior to scheduler call

Function Prototype:

```
void xTaskCreate(pvTaskCode,pcName, usStackDepth, pvParameters,
uxPriority, pxCreatedTask;
```

Function Parameters:

- **pvTaskCode:**Pointer to the function where the task is implemented
- **pcName:** Given name of the task. Intended for debugging purpose only
- **usStackDepth:** Length of the stack for this task in words
- **uvParameters:** Pointer to Parameter structure given to the task
- **uxPriority:** Priority given to the task, a number between 0 and MAX_PRIORITIES – 1 (see Kernel configuration)
- **pxCreatedTask:** Pointer to an identifier that allows handling the task. If the task does not have to be handled in the future, this can be NULL

3.2.2. xTaskDelete Function

To use this function, `INCLUDE_vTaskDelete` must be defined as 1 in `FreeRTOSConfig.h`. The `vTaskDelete` function is used to remove a task from the scheduler management (removed from all ready, blocked, suspended, and event lists). The identifier of the task to delete should be passed as parameter.

Function Prototype:

```
void vTaskDelete(xTaskHandlepxTask);
```

Function Parameters:

- **pxTask:** Pointer to identifier that allows handling the task to be deleted. Passing NULL will cause the calling task to be deleted.

Note: When a task is deleted, it is the responsibility of idle task to free all allocated memory to this task by kernel. Note that all memory dynamically allocated must be manually freed.Task deletion should be avoided in majority of RTOS apps in order to avoid HEAP actions, heavy in CPU cycles. It is preferable to have a task put in sleep mode and awakens on events for regular actions to obtain deterministic timings.

The following code example illustrates a simple task definition and creation:

```
#include <asf.h>
/* Task handler declaration*/
xTaskHandle worker1_id;


static void worker1_task(void *pvParameters)
{
    for(;;)
    {
        /* task application*/
    }
    /* Should never go there */
    vTaskDelete(NULL);
}

int main (void)
{
    Sysclk_init();
    Board_init();
    /* Create Worker 1 task */
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL, 2,& worker1_id);
    /*Start Scheduler*/
    vTaskStartScheduler()
```

```
    while(1);
}
```

TIP: As any code in infinite loop can fail and exit this loop, it is safer even for a repetitive task, to invoke `vTaskDelete()` before its final brace.

3.3. Task Management

FreeRTOS kernel offers various functions for task management. These functions allow setting tasks in different states and also obtain information on their status. The list of available functions are:

- vTaskDelay /* Delay a task for a set number of tick */
- vTaskDelayUntil /* Delay a task for a set number of tick */
- vTaskPrioritySet /* Set task priority */
- uxTaskPriorityGet /* Retrieve Task priority setting */
- vTaskSuspend /* Suspend a Task */
- vTaskResume /* Resume a Task */
- eTaskStateGet /* Retrieve the current status of a Task */
- vTaskDelete /* Delete a Task */

Most of these functions are used and described in the various examples in this document.

The following example illustrates the management of task:

```
#include <asf.h>
xTaskHandle worker1_id;
xTaskHandle worker2_id;

static void worker1_task(void *pvParameters)
{
    static uint32_t idelay;
    static uint32_t Delay;
    Delay = 100000;
    /* Worker task Loop. */
    for(;;)
    {
        /* Simulate work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
        vTaskSuspend(worker1_id);
    }
    /* Should never go there */
    vTaskDelete(worker1_id);
}
```

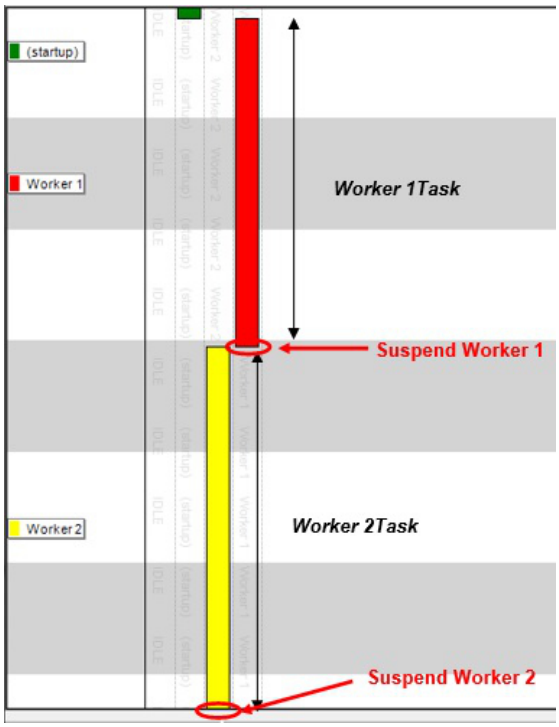
```
static void worker2_task(void *pvParameters)
{
    static uint32_t idelay;
    static uint32_t Delay;
    Delay = 100000;
    /* Worker task Loop. */
    for(;;)
    {

        /* Simulate CPU work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
```

```
        vTaskSuspend(worker2_id);
    }
    /* Should never go there */
    vTaskDelete(worker2_id);
}
```

```
int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL, 1,& worker1_id);
    /* Create Worker 2 task */
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL, 2,& worker2_id);
    /*Start Scheduler*/
    vTaskStartScheduler();
    while(1);
}
```

In this code example, two tasks with different priority are created. Each task simulates a CPU workload by performing a loop of a certain time (idelay). After the execution of this loop, the task is suspended using the `vTaskSuspend`. Since the task is not resumed in the program the tasks are executed only one time.

Figure 3-1 Illustration for Execution of Tasks



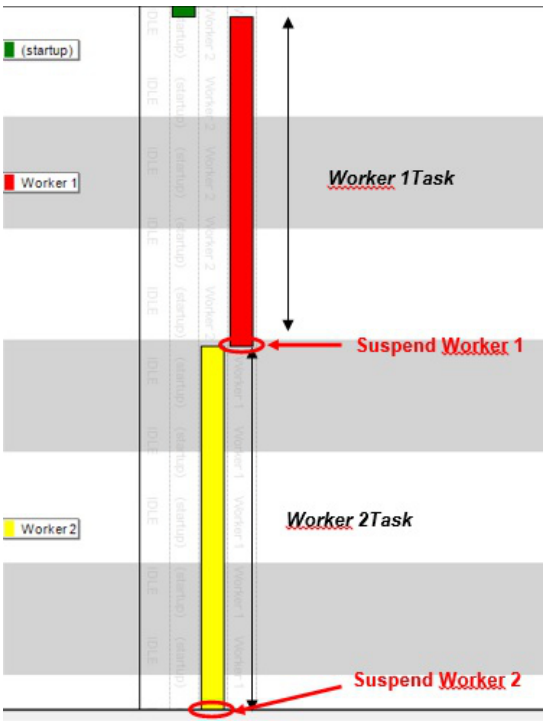
3.4. Priority Settings and Round Robin

FreeRTOS allows developer to set different level of priority for each task to be executed. The task priority setting is performed during task creation (xTaskCreate, uxPriority parameter). See the following snippet from the previous chapter example:

```
int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL, 1,& worker1_id);
    /* Create Worker 2 task */
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL, 2,& worker2_id);
    /*Start Scheduler*/
    vTaskStartScheduler();
    while(1);
}
```

Using various priority combinations will cause different impact on the tasks scheduling and execution. In the current example, the “worker 1 task has a higher priority than the worker 2 one. This results in the execution of “worker 1” task prior to “worker 2”.

Figure 3-2 Illustration for Execution of Tasks with Different Priority

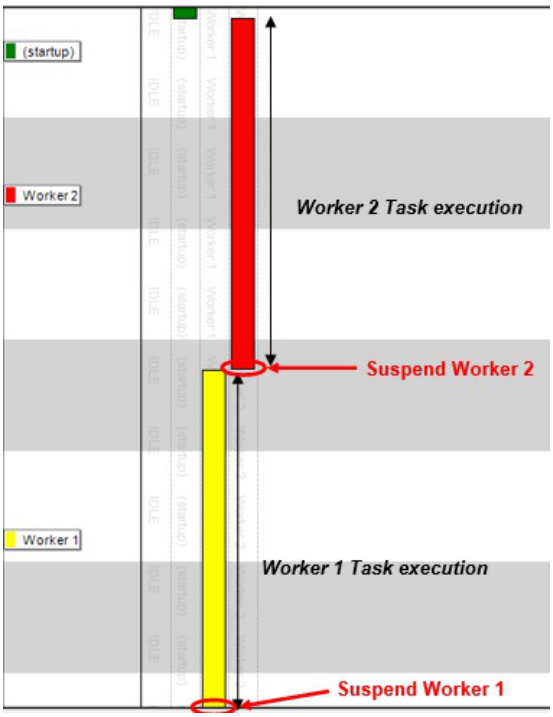


By modifying the code to set a higher priority on worker 2 task, allows to execute it prior to “worker 1” task:

```
int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL, 2,& worker1_id);
```

```
    /* Create Worker 2 task */
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL, 1,& worker2_id);
    /*Start Scheduler*/
    vTaskStartScheduler();
    while(1);
}
```

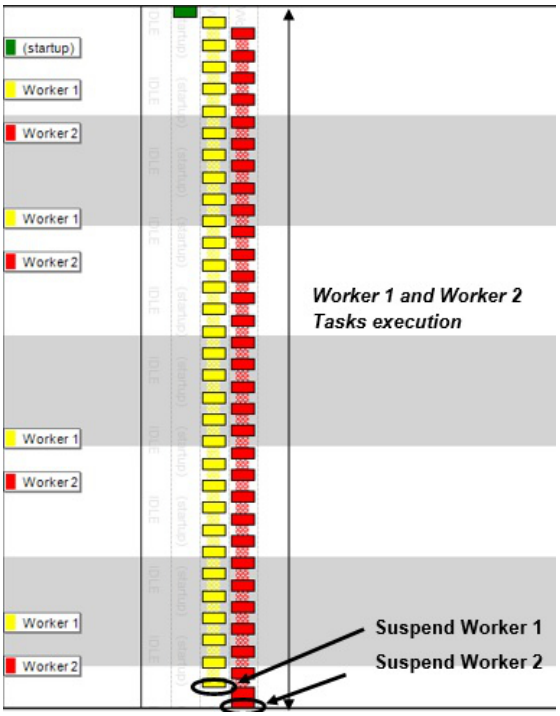
Figure 3-3 Illustration for Execution of Tasks with Switched Priority



When two or more tasks share the same priority, the scheduler will reduce their execution in time slice of one tick period and alternate their execution at each tick. This method of executing task is also called as round robin.

```
int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL, 1,& worker1_id);
    /* Create Worker 2 task */
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL, 1,& worker2_id);
    /*Start Scheduler*/
    vTaskStartScheduler();
    while(1);
}
```

Figure 3-4 Illustration for Execution of Tasks with Equal priority



4. Kernel Objects

In addition to the standard task scheduling and management functionality, FreeRTOS provides kernel objects that allow tasks to interact with each other. Upcoming sections describe the following standard kernel object.

- Software Timer
- Binary semaphores
- Queues

4.1. Software Timer Usage

A software timer allows a specific function to be executed at a preset duration in the future. The function executed by the timer is called the timer's callback function. The time between a timer being started and its callback function being executed, is called the timer's period. In short, the timer's callback function is executed when the timer period expires.

A timer can be linked to tasks using a specific handle ID. It also has a dedicated priority setting. Refer FreeRTOS config file.

```
xTimerHandle Timer_handle;
```

Different functions are used for creating and managing timers. Most of these functions require a timeout value of type `xBlockTime`. This timeout represents the maximum tick latency for the function to be taken into account. As the timer is like a task, it needs to have a higher priority, to be allowed to run when command is called. The `xBlockTime` is a time-out, in case the timer function is not handled on time. This provides the reason for this function to have one of the highest priorities in the system. A list of all these function are:

- **xTimerCreate:**
  Description: Function used to create a timer object

  Prototype: *xTimerHandle* xTimerCreate(*pcTimerName, xTimerPeriodInTicks, uxAutoReload, pvTimerID, pxCallbackFunction);

  Parameters: **pcTimerName**: Given name to the timer, for debugging purpose only

  xTimerPeriodInTicks: Number of tick in timer period

  uxAutoReload: If set to 1, activate timer auto reload feature

  pvTimerID: Pointer to pre defined timer ID (xTimerHandle)

  pxCallbackFunction: Pointer to callback function to be executed when the timer's period expires
- **xTimerStart:**
  Description: Function used to start a timer

  Prototype: void xTimerStart(xTimer, xBlockTime)

  Parameters: **xTimer**: targeted timer ID

  xBlockTime: Timeout for function to be handled by timer object
- **xTimerStop:**
  Description: Function used to stop a timer

  Prototype: void xTimerStop(xTimer, xBlockTime)

Parameters: **xTimer**: targeted timer ID

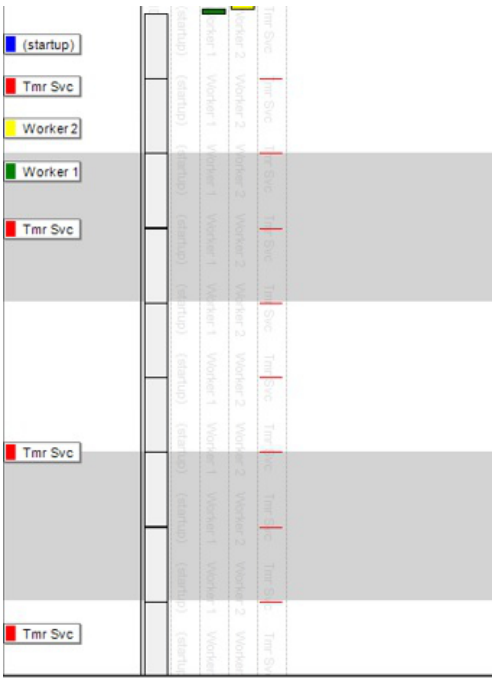
xBlockTime: Timeout for function to be handled by timer object

Returning to the contextual example, the inclusion of code formatted in bold in the following example, illustrates the process to initialize a 500 ticks software timer.

```
        #include <asf.h>
        xTaskHandle worker1_id;
        xTaskHandle worker2_id;
        xTimerHandle Timer_id;

        static void worker1_task(void *pvParameters)
        {
            static uint32_t idelay;
            static uint32_t Delay ;
            Delay = 100000;
            for(;;)
            {
                /* Simulate work */
                for (idelay = 0; idelay < Delay; ++idelay);
                /* Suspend Task */
                vTaskSuspend(worker1_id);
            }
            /* Should never go there */
            vTaskDelete(NULL);
        }
        static void worker2_task(void *pvParameters)
        {
            static uint32_t idelay;
            static uint32_t Delay ;
            Delay = 100000;
            for(;;)
            {
            /* Simulate CPU work */
            for (idelay = 0; idelay < Delay; ++idelay);
            /* Suspend Task */
            vTaskSuspend(worker2_id);
            }
            /* Should never go there */
            vTaskDelete(NULL);
        }
     void TimerCallback( xTimerHandle pxtimer )
        {
            /* Timer Callback section*/
        }

        int main (void)
        {
            board_init();
            sysclk_init();

            /*Create 2 Worker tasks. */
            xTaskCreate(worker1_task,"Worker1",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+1,&worker1_id);
            xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+2,&worker2_id);

         /* Create one Software Timer.*/
            Timer_id = xTimerCreate("Timer",500,pdTRUE,
0,TimerCallback);
```

```
/* Start Timer.*/
xTimerStart( Timer_id, 0);
vTaskStartScheduler();
```

Figure 4-1 Illustration for Execution of Tasks with Timer



4.2. Semaphore Usage

To synchronize different tasks together, FreeRTOS kernel provides semaphore objects. A semaphore can be compared to a synchronization token that tasks can exchange with each other.

To synchronize a task with an interrupt or another task, the task to synchronize will request a semaphore by using the function `xSemaphoreTake`. If the semaphore is not available, the task will be blocked waiting for its availability. At this time the CPU process will be released and another concurrent task that is ready will be able to start/continue its work. The task/interrupt to be synchronized with will have to execute `xSemaphoreGive` function in order to unblock the task. The task will then take the semaphore.

An example describing semaphore usage between hardware interrupt and task:

Figure 4-2 Semaphore not Available

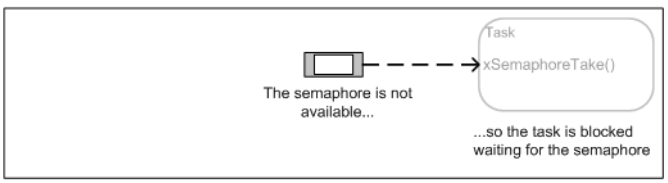


Figure 4-3 Interrupt (which gives semaphore) Occurs

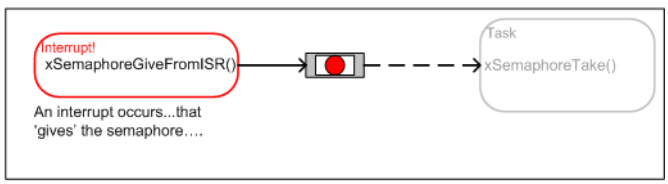


Figure 4-4 xSemaphoreGiveFromISR()

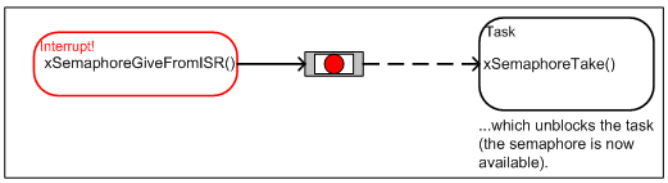
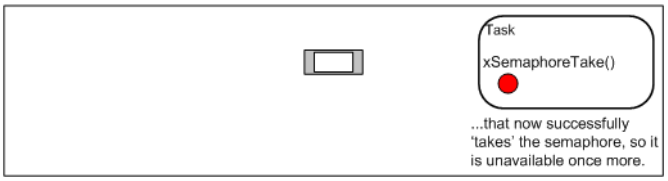


Figure 4-5 Semphore is Available



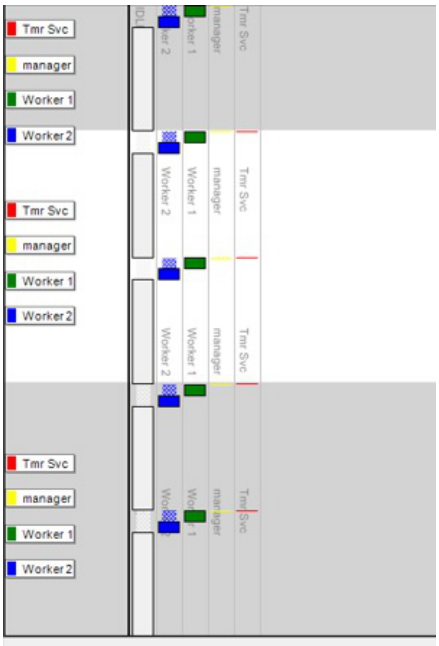
In the contextual example, the following hightlighted code illustrates the creation of a Manager Task that creates and uses a notification semaphore in order to synchronize its execution with the previously created timer. This Manager Task will have the highest priority in the system but will need the release of the notification semaphore (implemented in the timer callback) to be unblocked. This manager task function will be used to resume the worker task.

```
#include <asf.h>

xTaskHandle worker1_id;
xTaskHandle worker2_id;
xTaskHandle manager_id;
xTimerHandle Timer_id;
xSemaphoreHandle notification_semaphore;

static void worker1_task(void *pvParameters)
{
    static uint32_t idelay,Delay ;
    Delay = 100000;
    for(;;)
    {
        /* Simulate work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
        vTaskSuspend(worker1_id);
    }
    /* Should never go there */
    vTaskDelete(NULL);
}

static void worker2_task(void *pvParameters)
{
    static uint32_t idelay , Delay;
    Delay = 100000;
    for(;;)
    {
        /* Simulate CPU work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
        vTaskSuspend(worker2_id);
    }
    /* Should never go there */
    vTaskDelete(NULL);
}
```

```
void TimerCallback( xTimerHandle pxtimer )
{
    /* notify manager task to start working. */
    xSemaphoreGive(notification_semaphore);
}
```

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

    /* Create the notification semaphore and set the initial state. */
    vSemaphoreCreateBinary(notification_semaphore);
    vQueueAddToRegistry(notification_semaphore, "Notification Semaphore");
    xSemaphoreTake(notification_semaphore, 0);

    for(;;)
    {
        /* Try to get the notification semaphore. */
        /* The notification semaphore is only released in the SW Timer
callback */
        if (xSemaphoreTake(notification_semaphore, 10000))
        {
            vTaskResume(worker1_id);
            vTaskResume(worker2_id);
        }
    }
}


int main (void)
{
    board_init();
    sysclk_init();

    /*Create 2 Worker tasks. */
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+1,&worker1_id);
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+2,&worker2_id);

    /* Create one Software Timer.*/
    Timer_id = xTimerCreate("Timer",500,pdTRUE,0,TimerCallback);
    /* Start Timer.*/
    xTimerStart( Timer_id, 0);

 /* Create one manager task.*/
    xTaskCreate(manager_task,"manager",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+3,&manager_id);

    vTaskStartScheduler();
    while(1);
}
```

Figure 4-6 Illustration for Execution of Tasks with Semaphore

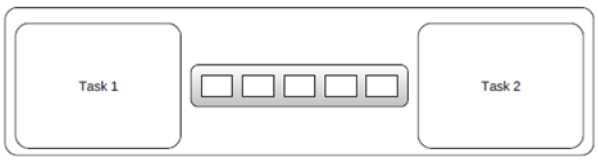


4.3. Queue Management

Queues are used for inter-task communication and synchronization in a FreeRTOS environment. They are an important subject to understand as it is unavoidable to be able to build a complex application with tasks interacting with each other. They are meant to store a finite number of fixed size data. Queues should be accessible for reads and writes by several different tasks and do not belong to any task in particular. A queue is normally a FIFO which means elements are read in the order they have been written. This behavior depends on the writing method: Two writing functions can be used to write either at the beginning or at the end of this queue.

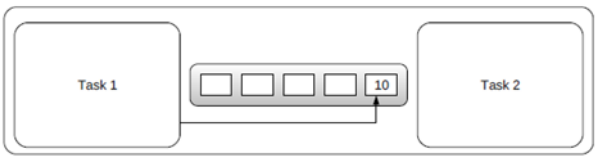
Illustration of Queue Usage: A Queue is created to allow task 1 and task 2 to communicate. The queue can hold a maximum of five values. When a queue is created, it does not contain any value. It is empty:

Figure 4-7 Queue is Empty



Task 1 writes a value on the queue; the value is sent to the end. Since the queue was previously empty, the value is now both the first and the last value in the queue:

Figure 4-8 Task1 Write First Value to the Queue



Task 1 sends another value. The queue now contains the previously written value and this newly added value. The previous value remains at the front of the queue while the new one value is placed after the first value. Three spaces are still available:

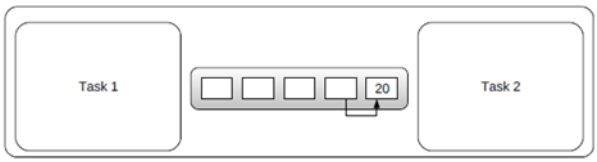
Figure 4-9 Task1 Write another Value to Queue

Task 2 reads a value in the queue. It will receive the value in the front of the queue:

Figure 4-10 Task2 Reads a Value

Task 2 has removed an item. The second item is moved to be the one in the front of the queue. This is the value task 2 will read next time it tries to read a value. Four spaces are now available:

Figure 4-11 Second Value Moved to Front in Queue



The list of the kernel functions that allows to handle the queue are:

- **xQueueCreate:**
  Description: Function used to create a new queue

  Prototype: xQueueCreate(uxQueueLength, uxItemSize);

  Parameters: **uxQueueLength**: Number of item that queue can store

  uxItemSize: Size of the item to be stored in queue
- **xQueueSend:**
  Description: Function used to send data into a queue

  Prototype: xQueueSend(xQueue, pvItemToQueue, xTicksToWait)

  Parameters: **xQueue**: ID of the Queue to send data in

  pvItemToQueue: Pointer to Data to send into Queue

  xTicksToWait: System wait for command to be executed
- **xQueueReceive:**
  Description: Function used to receive data from a queue

  Prototype: xQueueReceive(xQueue, pvBuffer, xTicksToWait)

  Parameters: **xQueue**: ID of the Queue to send data in

  pvItemToQueue: Pointer to Data to send into Queue

  xTicksToWait: System wait for command to be executed

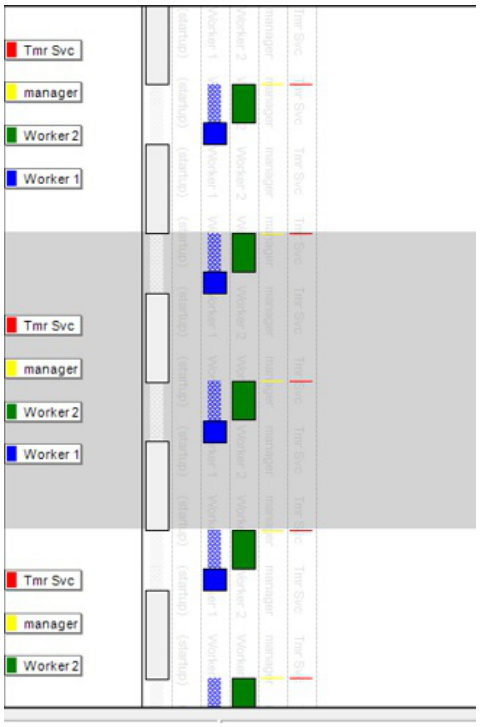
The following contextual example illustrates the queue usage by passing CPU load workload simulation information (delay) from manager to Worker task using a message queue. Refer to the code dispalyed in bold format.

```
#include <asf.h>

xTaskHandle worker1_id;
xTaskHandle worker2_id;
xTaskHandle manager_id;
xTimerHandle Timer_id;
xSemaphoreHandle notification_semaphore;
xQueueHandle Queue_id;

static void worker1_task(void *pvParameters)
{
    static uint32_t idelay,Delay;
       xQueueReceive(Queue_id,&Delay,100000);
    for(;;)
    {
        /* Simulate work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
        vTaskSuspend(worker1_id);
    }
    /* Should never go there */
    vTaskDelete(NULL);
}

static void worker2_task(void *pvParameters)
{
    static uint32_t idelay , Delay;
       xQueueReceive(Queue_id,&Delay,100000);
    for(;;)
    {
        /* Simulate CPU work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task */
        vTaskSuspend(worker2_id);
    }
    /* Should never go there */
    vTaskDelete(NULL);
}

void TimerCallback( xTimerHandle pxtimer )
{
    /* notify manager task to start working. */
    xSemaphoreGive(notification_semaphore);
}
```

```
static void manager_task(void *pvParameters)
{
 static uint32_t Delay1 = 400000 , Delay2 = 200000;

    /* Create the notification semaphore and set the initial state */
    vSemaphoreCreateBinary(notification_semaphore);
    vQueueAddToRegistry(notification_semaphore, "Notification Semaphore");
    xSemaphoreTake(notification_semaphore, 0);
```

```
    for(;;)
    {
        /* Try to get the notification semaphore. */
        /* The notification semaphore is only released in the SW Timer
callback */
        if (xSemaphoreTake(notification_semaphore, 10000))
        {
         xQueueSend(Queue_id,&Delay1,0);
            xQueueSend(Queue_id,&Delay2,0);
            vTaskResume(worker1_id);
            vTaskResume(worker2_id);
        }
    }
}


int main (void)
{
    board_init();
    sysclk_init();

    /*Create 2 Worker tasks */
    xTaskCreate(worker1_task,"Worker 1",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+1,
    &worker1_id);
    xTaskCreate(worker2_task,"Worker 2",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+2,
    &worker2_id);

    /* Create one Software Timer */
    Timer_id = xTimerCreate("Timer",500,pdTRUE,0,TimerCallback);
    /* Start Timer.*/
    xTimerStart( Timer_id, 0);

    /* Create one manager task */
    xTaskCreate(manager_task,"manager",configMINIMAL_STACK_SIZE
+1000,NULL,tskIDLE_PRIORITY+3,
    &manager_id);

 /* Create a queue*/
    Queue_id = xQueueCreate(2,sizeof( unsigned long ));

    vTaskStartScheduler();
    while(1);
}
```

Figure 4-12 Illustration for Execution of Taks with Queue



5. Hook Functions

In addition to task management functions, FreeRTOS provides hook functions that allow management of additional event in the system. Hook functions (or callbacks) are called by the kernel when specific predefined event appends. The usage of each hook can be disabled or enable from the FreeRTOS configuration file.

5.1. Idle Hook Function

The idle task (default lowest priority task) can call an application defined hook function - the idle hook. This function will be executed only when there are no tasks of higher priority that must be executed. Hence, idle hook function an ideal state to place the processor into a low power state - providing an automatic power saving whenever no processing is required.

The idle hook is called repeatedly as long as the idle task is running.

WARNING: It is important that the idle hook function does not call any API functions that could cause it to block. If the application makes use of the `vTaskDelete()` API function then the idle task hook must be allowed to periodically return. This is because the idle task is responsible for cleaning up the resources that were allocated by the RTOS kernel to the task that has been deleted.

5.2. Tick Hook Function

The tick interrupt can optionally call the tick hook. The tick hook provides a convenient place to implement timer functionality.

WARNING: The `vApplicationTickHook()` function executes from within an ISR. It must be very short. It should not use much stack nor call any API functions that does not end in `FromISR` or `FROM_ISR`.

5.3. Malloc Failed Hook Function

The memory allocation schemes implemented by `heap_1.c`, `heap_2.c`, `heap_3.c`, and `heap_4.c` can optionally include a `malloc()` failure hook (or callback) function that can be configured to get called if `pvPortMalloc()` ever returns NULL.

Defining the `malloc()` failure hook will help identify problems caused by lack of heap memory – especially when a call to `pvPortMalloc()` fails within an API function.

WARNING: Dynamic memory allocation is not the best to do on MCUs. In the case of FreeRTOS, it is preferable to use dynamic memory allocation only during initialization.

6. References

1. FreeRTOS latest version Download link: http://www.freertos.org/a00104.html

7. Revision History

Doc. Rev.	Date	Comments
42622A	11/2015	Initial document release.

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**