

---

**AT04056: Getting Started with FreeRTOS on Atmel SAM  
Flash MCUs**

---

**Atmel | SMART**

---

**Introduction**

---

This application note illustrates the basic functionality of the FreeRTOS™ Real Time Operating System and show how to use it on SAM microcontroller by covering following points:

- What are 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
- How to make use of Graphical debugging tool

The description is based on FreeRTOS kernel port available in Atmel Software Framework (ASF). All the processes illustrated in this document can be reproduced on any Atmel Studio project for SAM devices based on ASF3.8.1 or higher.

## Table of Contents

---

<b>1</b>	<b>Introduction .....</b>	<b>3</b>
1.1	What is a Real-Time Application? .....	3
1.2	Real-Time Operating System and Multitasking .....	3
1.3	FreeRTOS Introduction .....	3
1.3.1	The FreeRTOS Kernel.....	4
1.3.2	FreeRTOS Tasks Management Mechanism.....	5
1.3.3	FreeRTOS Memory Management .....	6
<b>2</b>	<b>FreeRTOS Kernel Inclusion and Configuration .....</b>	<b>6</b>
2.1	Add the Kernel to an Existing Project.....	7
2.2	Configuration the Kernel According to Application Requirement.....	9
2.2.2	System and Tick Frequency .....	11
<b>3</b>	<b>Tasks Creation and Scheduling .....</b>	<b>12</b>
3.1	Task Structure .....	12
3.2	Task Creation and Deletion.....	13
3.2.1	XTaskCreate Function.....	13
3.2.2	VTaskDelete Function .....	13
3.3	Task Management.....	14
3.4	Priority Settings and Round Robin .....	17
<b>4</b>	<b>Kernel Objects .....</b>	<b>20</b>
4.1	Software Timer Usage.....	20
4.2	Semaphore Usage .....	23
4.3	Queue Management.....	27
<b>5</b>	<b>Hook Functions .....</b>	<b>32</b>
5.1	Idle Hook Function .....	32
5.2	Tick Hook Function.....	33
5.3	Malloc Failed Hook Function .....	33
<b>6</b>	<b>Debugging a FreeRTOS Application .....</b>	<b>33</b>
6.1	FreeRTOS+Trace Integration.....	35
6.2	Debug your Application using FreeRTOS+Trace .....	37
<b>7</b>	<b>Revision History .....</b>	<b>41</b>

# 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.

In order to fit with these time requirements, the usage of a real time operating system (RTOS) is often needed.

## 1.2 Real-Time Operating System and Multitasking

The most basic feature, common to all operating system is the support for multitasking. On top of this, can be added the support of networking, peripheral interfacing, user interface, printing, etc...

An embedded system may not require all of this, but need some of them. 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 300 bytes 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 in such way that all the deadlines (requirements) are met.

A real time operating system will always include the following features:

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

## 1.3 FreeRTOS Introduction

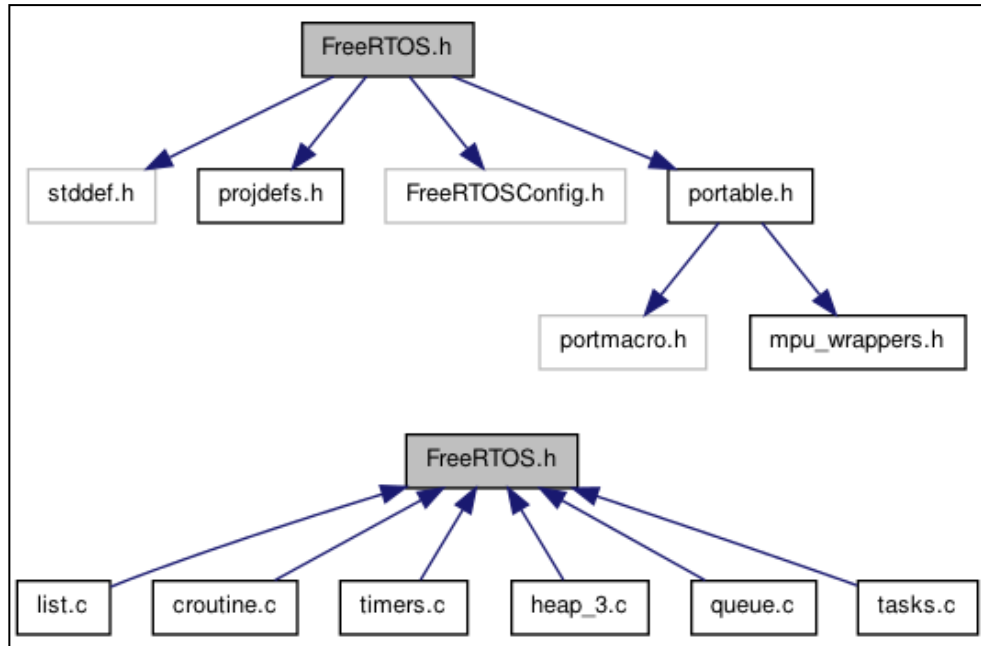
FreeRTOS is a real-time kernel (or real-time scheduler) on top of which Cortex®-M3/M4 microcontroller applications can be built to meet their hard real-time requirements. It allows Cortex-M3/M4 microcontroller applications to be organized as a collection of independent tasks to be executed. As most Cortex-M3/M4 microcontrollers 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 and is distributed as an independent module under the Atmel Software Framework. This module can be added in any standard project using the ASF wizard available under Atmel Studio or can be added manually when using the standalone version of ASF under IAR™.

**Figure 1-1. The FreeRTOS Module is Made of the Following Source Files**



The Cortex-M3/M4/M0+ 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

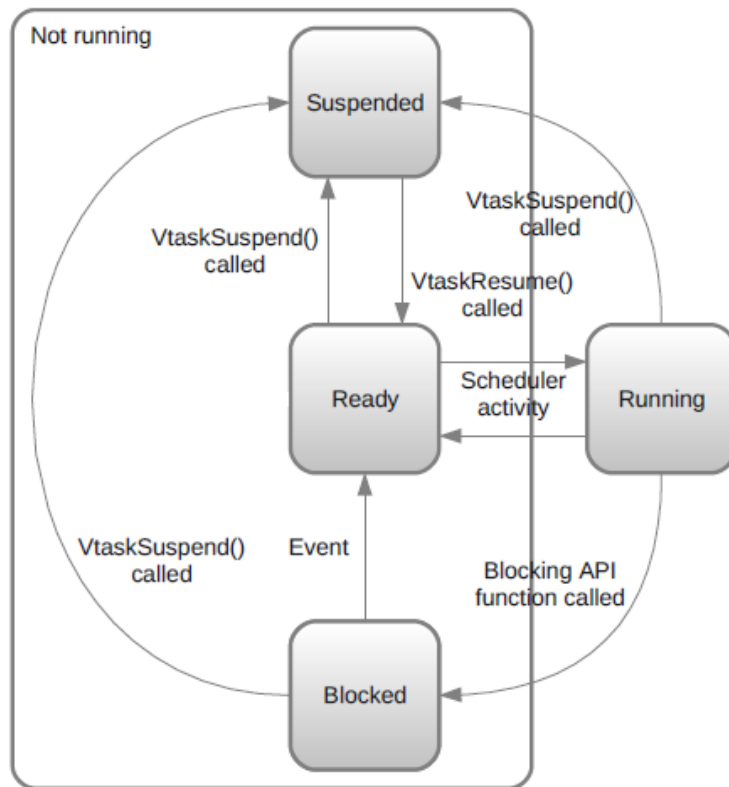
FreeRTOS can be configured to exclude unused functionality from compiling and so reduce its memory footprint.

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 Tasks 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 different task states are illustrated in Figure 1-2.

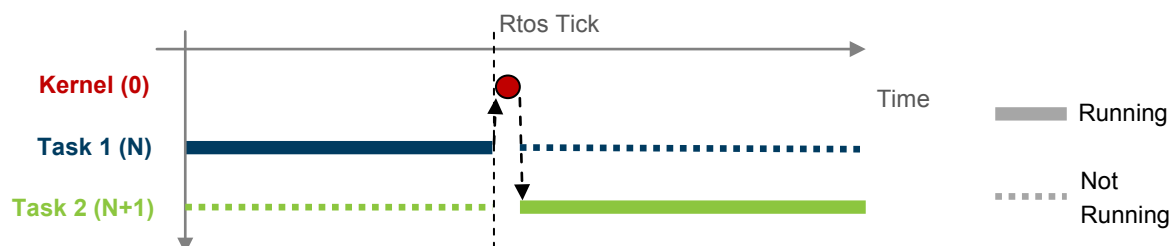
Figure 1-2. Illustrates the Different Tasks of the FreeRTOS Kernel



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

- **Suspend**: Task has been suspended (deactivated) by the application
- **Blocked**: Task is blocked and waiting for synchronization event
- **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.

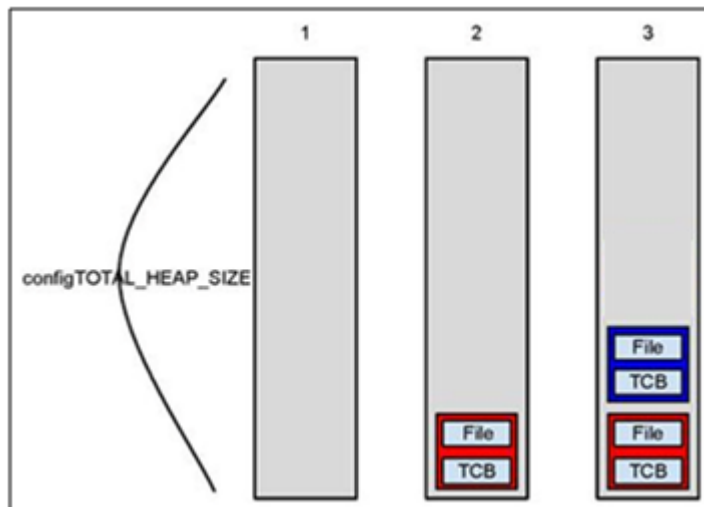


### 1.3.3 FreeRTOS Memory Management

FreeRTOS allows an unlimited number of tasks to be run 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.

Figure 1-3 illustrates the memory allocation of tasks in RAM.

Figure 1-3. Illustration of 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 to 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).

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

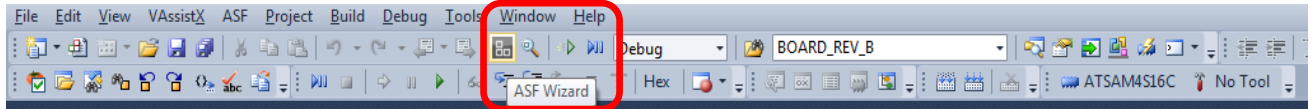
This chapter describes the inclusion and configuration of FreeRTOS kernel in an Atmel studio 6.1 project. By covering the following aspects:

- Use the ASF wizard to add the kernel in an existing project
- Configure FreeRTOS kernel according to product specification
- Optimize kernel size according to application requirements

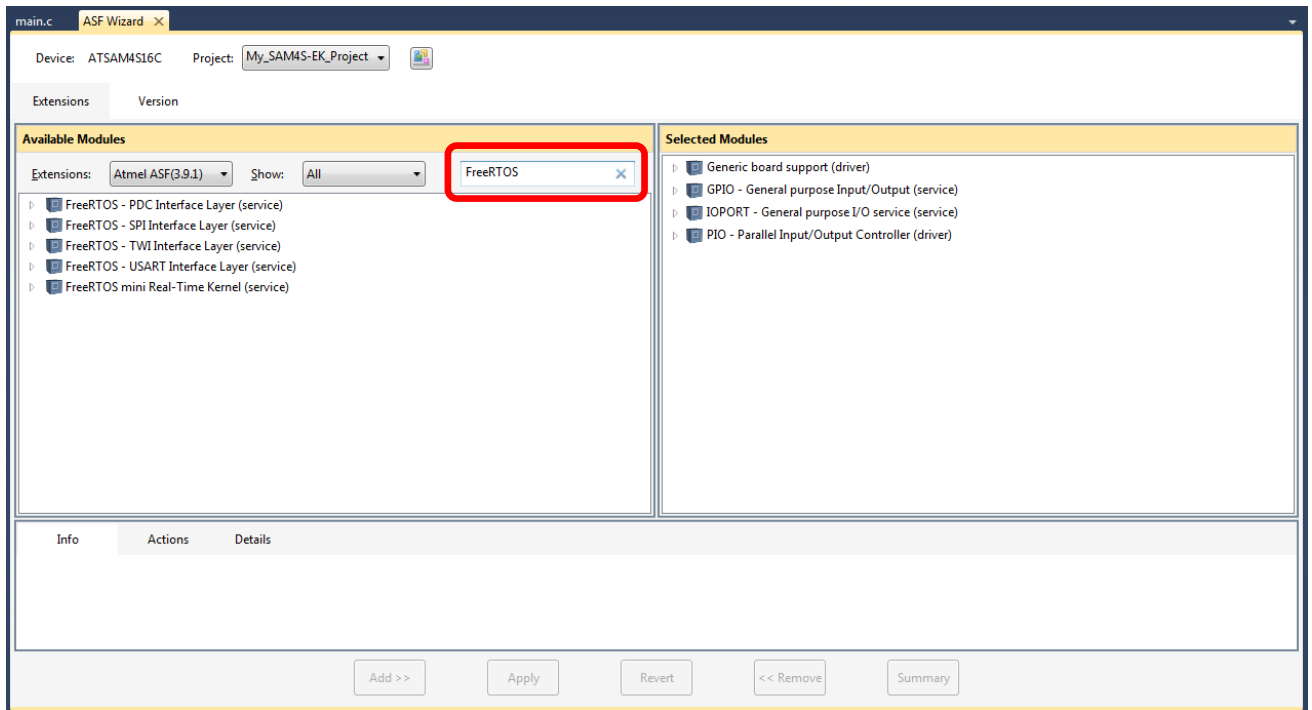
## 2.1 Add the Kernel to an Existing Project

The FreeRTOS kernel is available in the Atmel Software Framework (ASF) as a standard module that can be added to any project built around ASF. In this document the process is illustrated in an SAM4S-Xplained Pro project, but is applicable for all existing project.

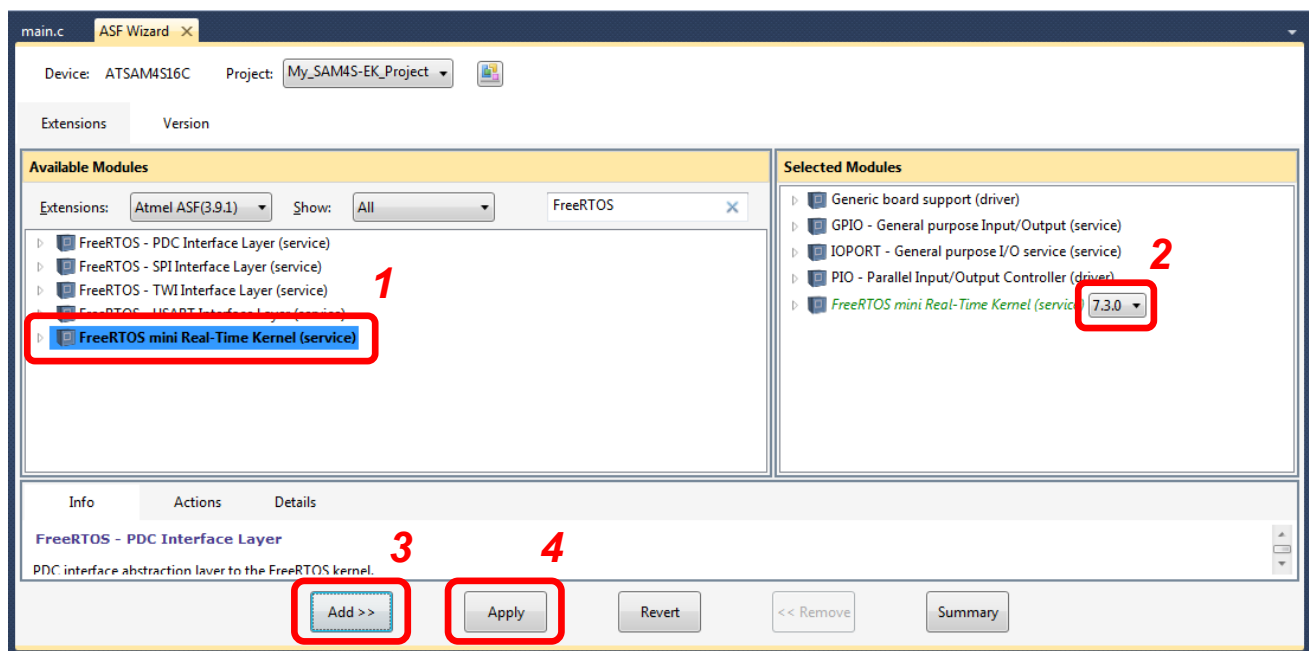
In any existing project, the kernel module addition is performed through the ASF wizard of Atmel Studio 6.1.



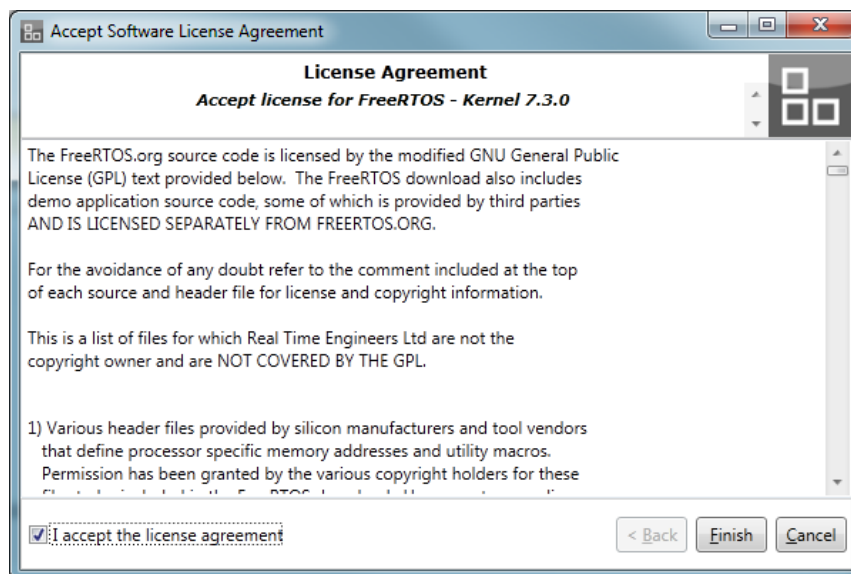
Different FreeRTOS modules are available in the wizard, all of them start with “FreeRTOS” and can be easily found by using the search bar from the “ASF wizard window”.



The kernel is an independent module called “**FreeRTOS mini real time kernel**”. When selected in the available module list, that can be add by clicking on “**Add >>**” button then “**Apply**” button. During the selection you can specify which version of the kernel to include in the project.

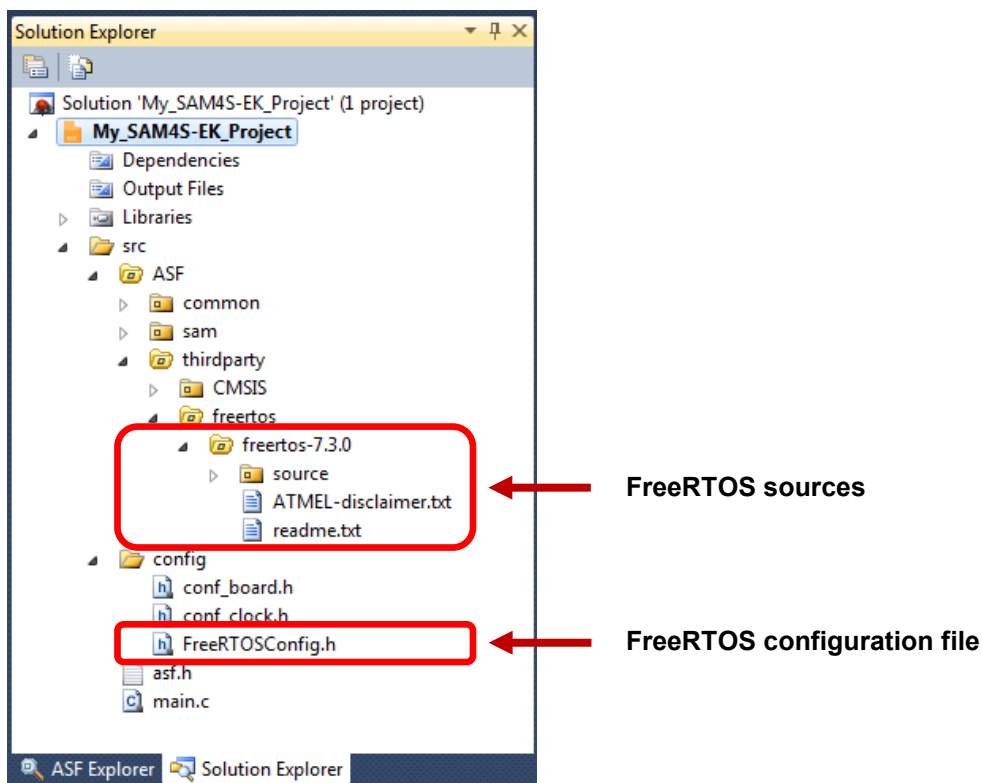


Note: FreeRTOS source code is licensed by the modified GNU General Public License (GPL). A license agreement is then required to use the kernel in an industrial project. Information on this license can be found in the license agreement window that appears when adding the kernel.



After module addition the kernel sources and configuration files are available in the solution explorer under “**src/ASF/thirdparty/FreeRTOS**” path.

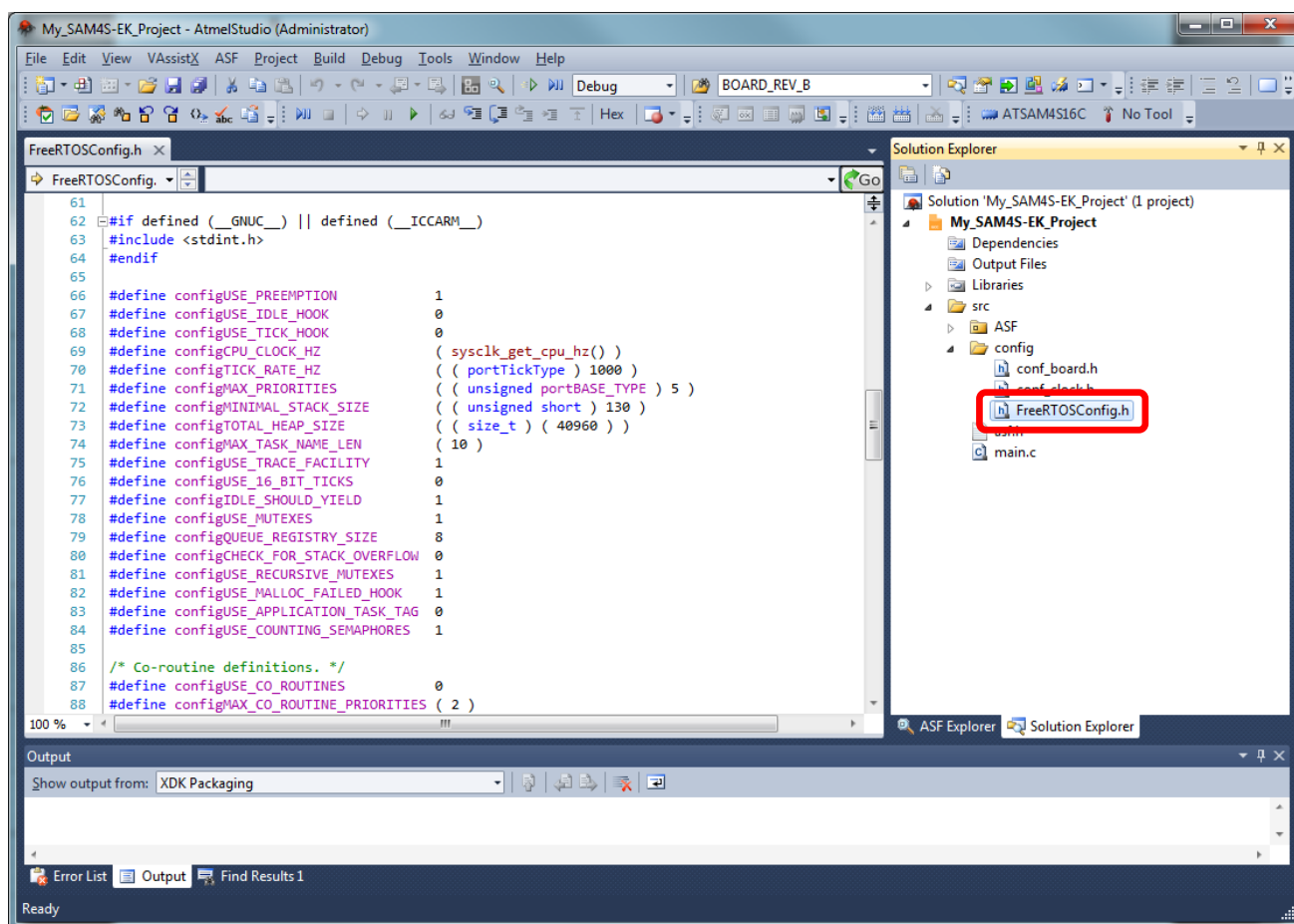




Note: In addition of the inclusion of source in the project, the wizard automatically adds FreeRTOS module sources path to the compiler input directory path.

## 2.2 Configuration the Kernel According to Application Requirement

The kernel configuration is done through a dedicated header files ("FreeRTOSConfig.h") available in the project conf directory see following figure. The kernel configuration is achieved by modifying some predefine "config" and "INCLUDE" definitions. By default these definition are already set.



Taking time to adapt the kernel to application needs allows reducing the footprint of the kernel in memory. [Table 2-1](#) lists the FreeRTOS kernel configuration and customization definitions that can be found in the FreeRTOSConfig.h.

**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
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)

Config definition	Description
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

### 2.2.2 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:

```
#define configCPU_CLOCK_HZ          (sysclk_get_cpu_hz())
#define configTICK_RATE_HZ         ((portTickType)1000)
```

The first definition uses the “sysclk\_get\_cpu\_hz” function from ASF to retrieve the frequency of the CPU. In a project based on ASF, the CPU frequency is set according “conf\_clock.h” file.

The second definition allows setting the tick frequency in Hz.



In order to ensure that sysclk\_get\_cpu\_hz() return the correct system frequency, the “sysclk\_init” function should be added at the beginning of the main routine, to ensure that correct frequency configuration is applied during code execution.

```
#include <asf.h>
```

```
Int main (void)
```

```
{
```

```
    Sysclk_init();
```

```
    Board_init();
```

```
    ...
```

```
}
```

## 3 Tasks Creation and Scheduling

This chapter goes through the basic tasks creation, scheduling, and handling processes by describing:

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

All steps will be illustrated by FreeRTOS+Trace in order to see the impact of the different kernel function calls and settings. For more detail on how to configure “FreeRTOS+Trace” refer to Chapter 6 [Debugging a FreeRTOS Application](#).

### 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 below:

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

As no return is performed, 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;
```

In order to be executed, a task need to be created (Memory allocation + add to Scheduling list). At creation, a handler ID is assigned to each task. This ID will be 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 done by kernel function “**xTaskCreate()**” and “**vTaskDelete()**”.

### 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 its creation a task is ready to be executed. **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
- PvParameters: 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 VTaskDelete Function

In order to use the “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.

Prototype:

```
void vTaskDelete(xTaskHandle pxTask);
```

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);
}

```



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 different functions for task management. These functions allow setting tasks in different states and also obtain information on their status. Here is a list of available functions:

<code>VTaskDelay :</code>	<i>/* Delay a task for a set number of tick */</i>
<code>VTaskDelayUntil</code>	<i>/* Delay a task for a set number of tick */</i>
<code>VTaskPrioritySet</code>	<i>/* Set task priority */</i>
<code>UxTaskPriorityGet</code>	<i>/* Retrieve Task priority setting */</i>
<code>VTaskSuspend</code>	<i>/* Suspend a Task */</i>
<code>VTaskResume</code>	<i>/* Resume a Task */</i>
<code>ETaskStateGet</code>	<i>/* Retrieve the current status of a Task */</i>
<code>VTaskDelete</code>	<i>/* Delete a Task */</i>

Most of these functions will be used and described in the different example of the document.

To illustrate the management of task and also following description on kernel object usage we will take the following context example:

```

#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);
}

```

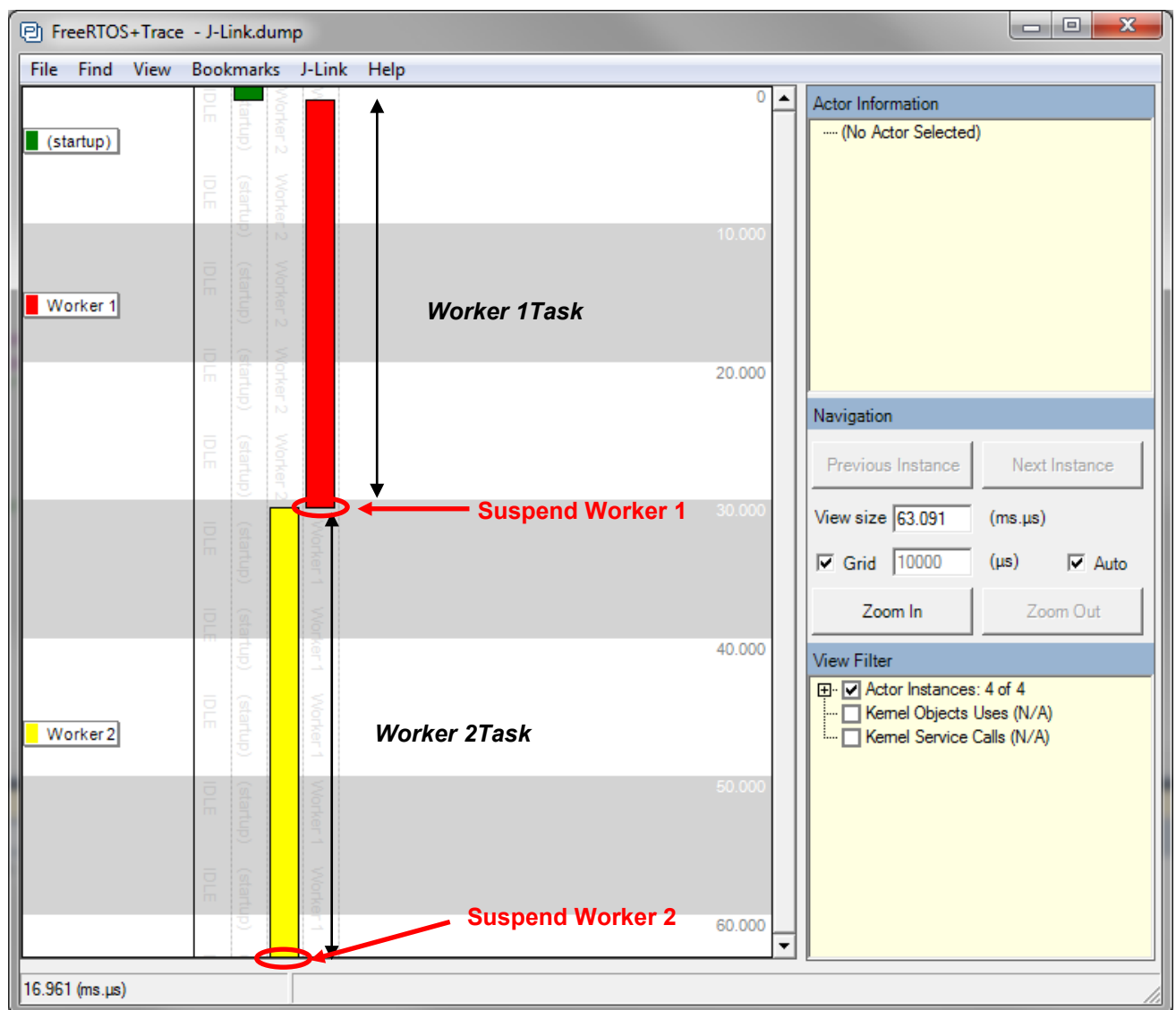
```

Int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task, "Worker 1", configMINIMAL_STACK_SIZE+1000, NULL, 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);
}

```

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”. As they are not resumed in the program the tasks are executed only one time.

**Figure 3-1. Result on FreeRTOS+Trace**



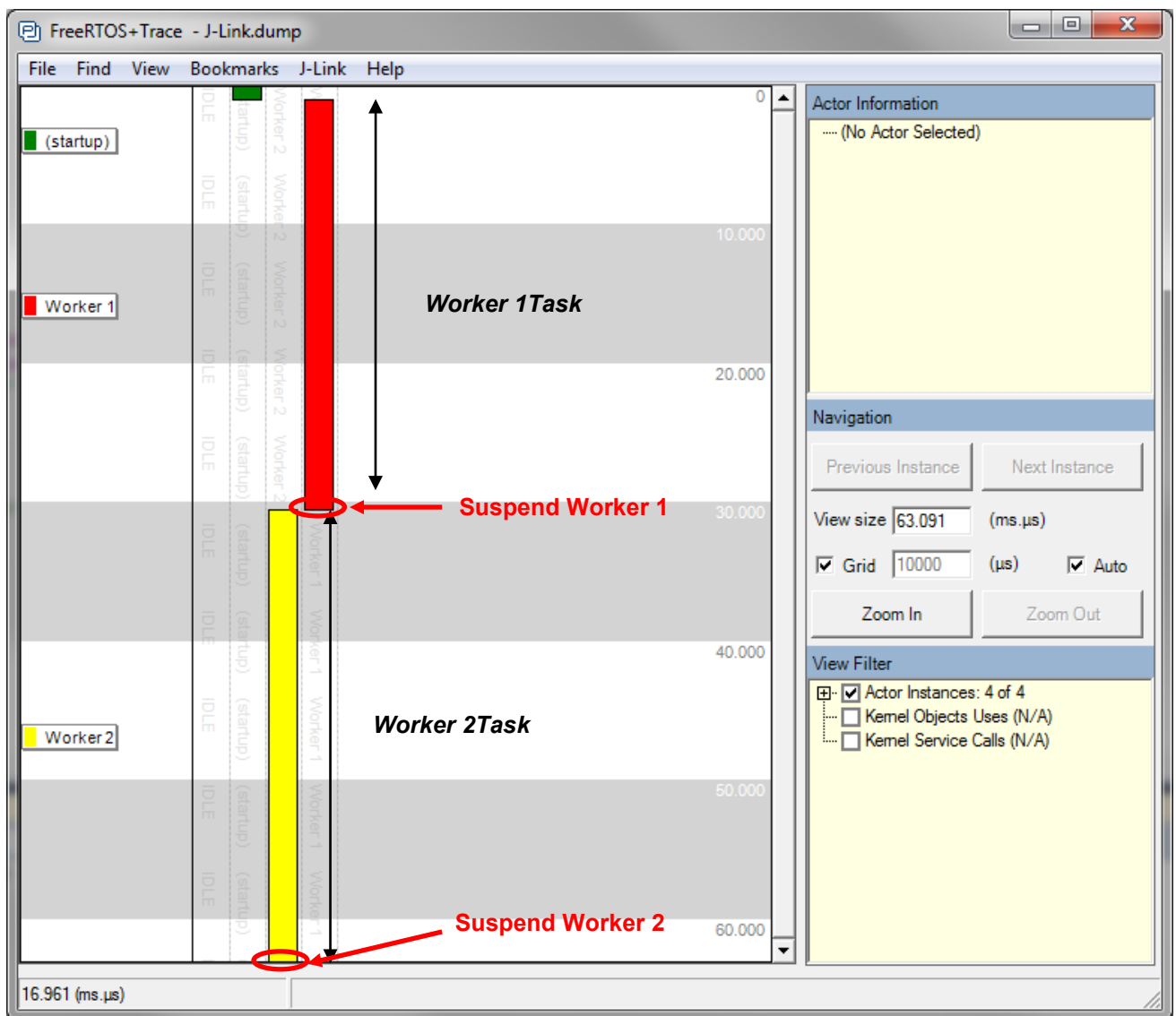


### 3.4 Priority Settings and Round Robin

FreeRTOS allows developer to affect 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 extract from the previous chapter example:

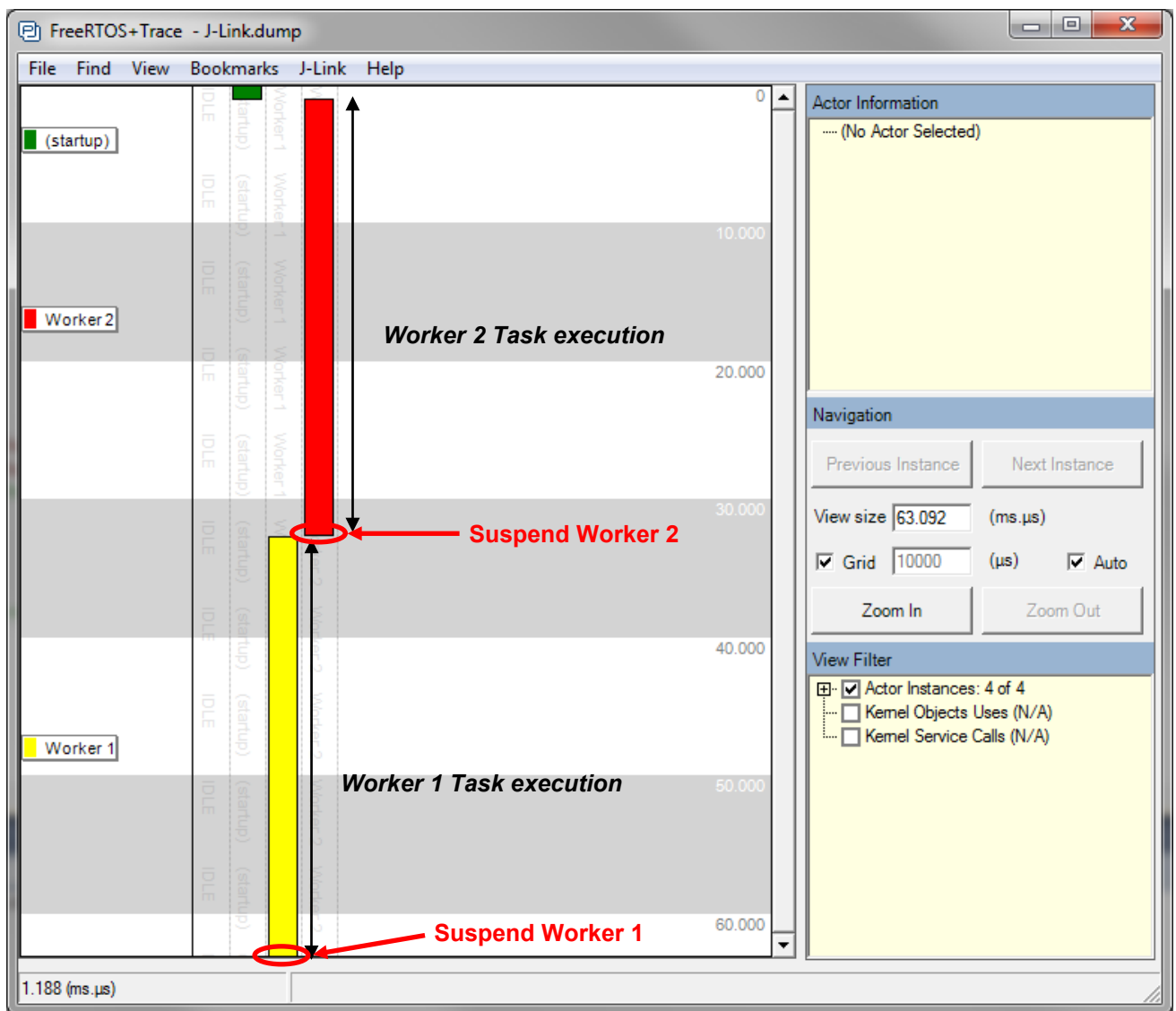
```
Int main (void)
{
    Sysclk_init();
    Board_init();
    xTaskCreate(worker1_task, "Worker 1", configMINIMAL_STACK_SIZE+1000, NULL, 2, &worker1_id);
    /* Create Worker 2 task */
    xTaskCreate(worker2_task, "Worker 2", configMINIMAL_STACK_SIZE+1000, NULL, 1, &worker2_id);
    /*Start Scheduler*/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

Using different priority combination will have a 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”.



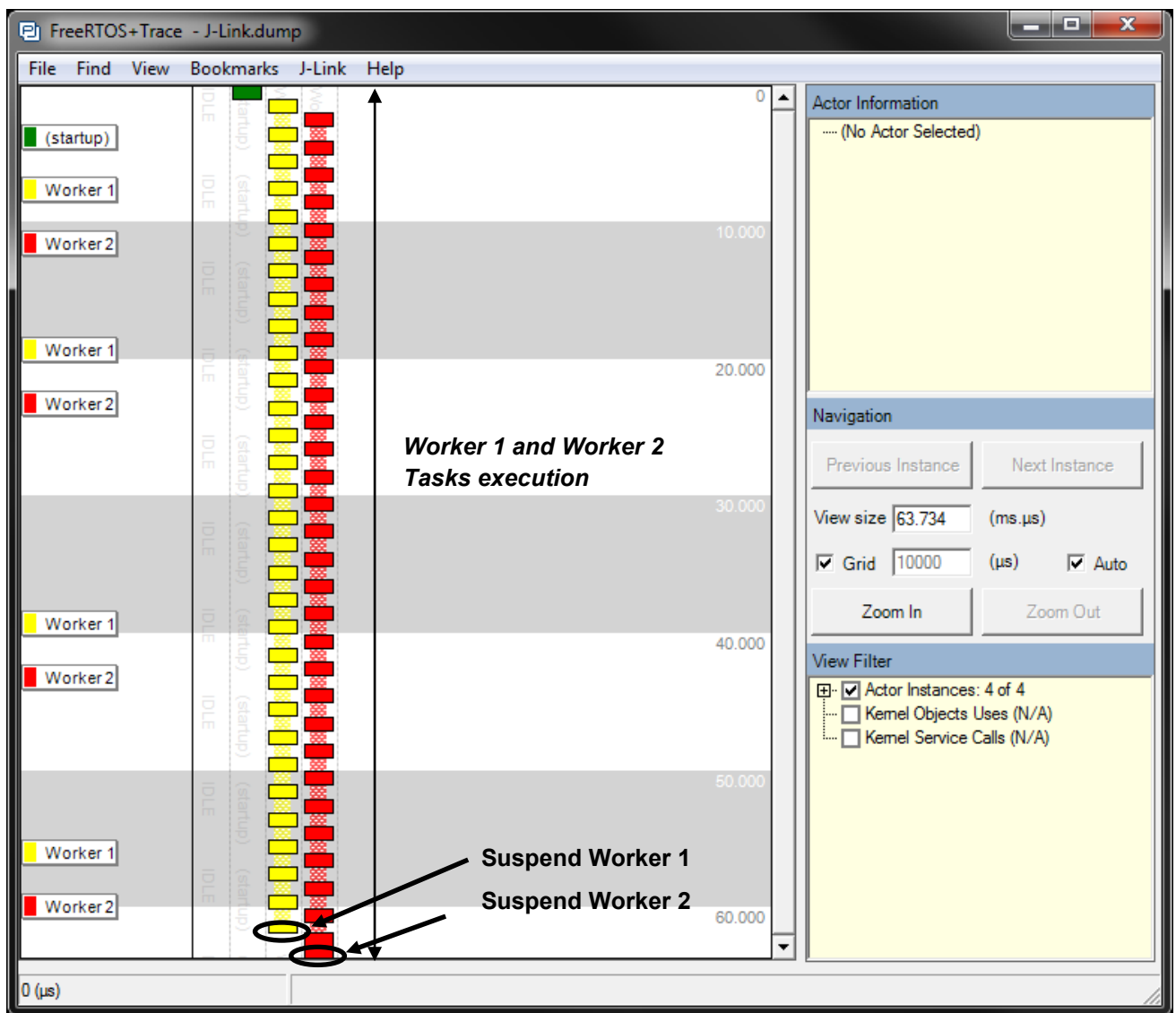
By modifying the code to set a higher priority on worker 2 task, will allow to execute it prior to “worker 1” task:

```
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);
}
```



When two or more tasks share the same priority, the scheduler will cut their execution in time slice of one tick period and alternate their execution at each tick. This way of executing task is also known 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);
}
```



## 4 Kernel Objects

In addition to standard task scheduling and management functionality, FreeRTOS provides kernel objects that allow tasks to interact each other. In this chapter we will focus on 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 set time 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 (see FreeRTOS config file).

```
xTimerHandle Timer_handle;
```

Different functions are used for creating and managing timers. Most of these functions need 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 is why it should have one of the highest priorities in the system. Here is a list of all these function:

- **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

Going back to the contextual example, the inclusion of code highlighted in red 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*/
}

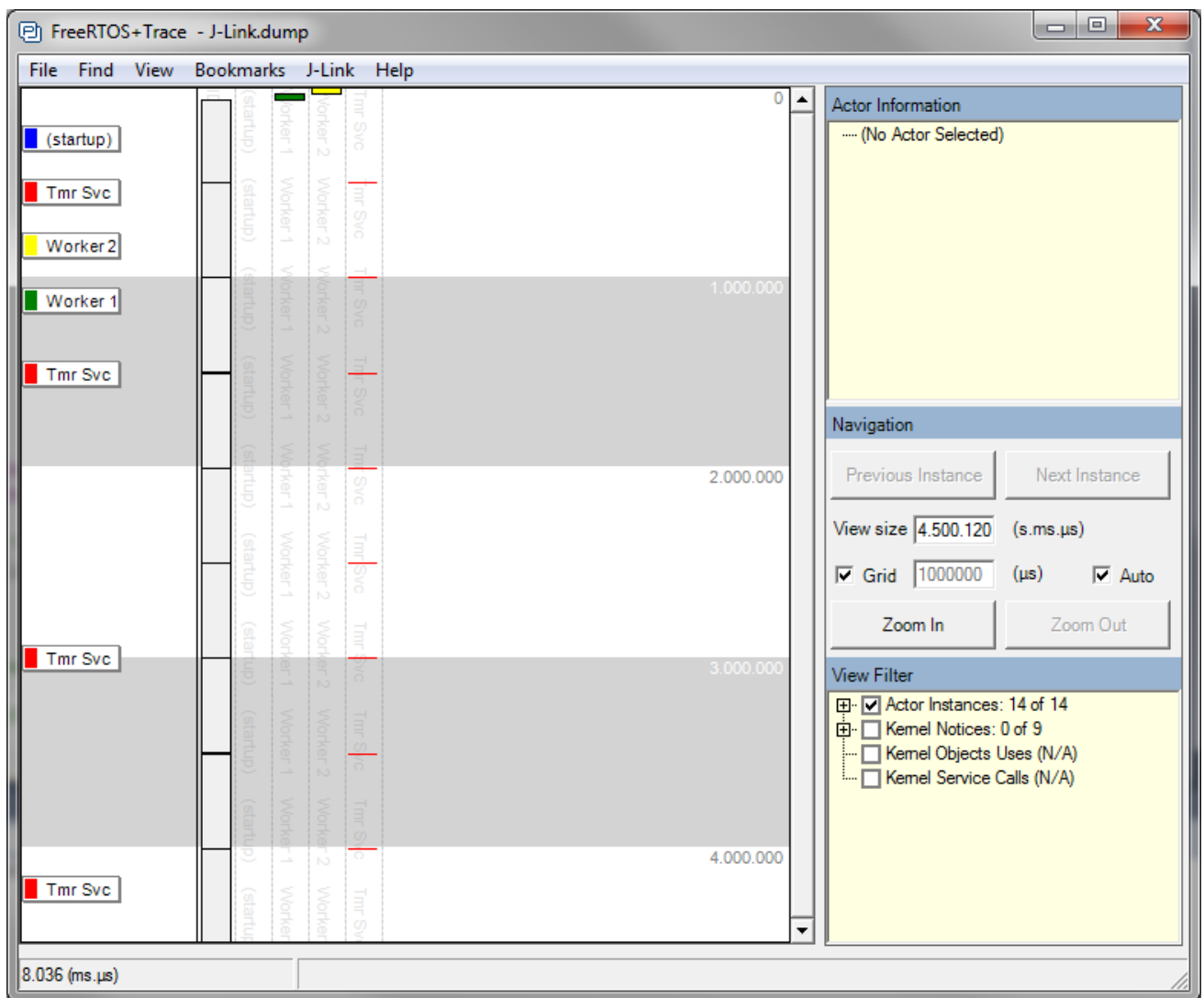
int main (void)
{
    board_init();
    sysclk_init();

    /*Create 2 Worker tasks. */
    xTaskCreate(worker1_task,"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);

    vTaskStartScheduler();
}

```

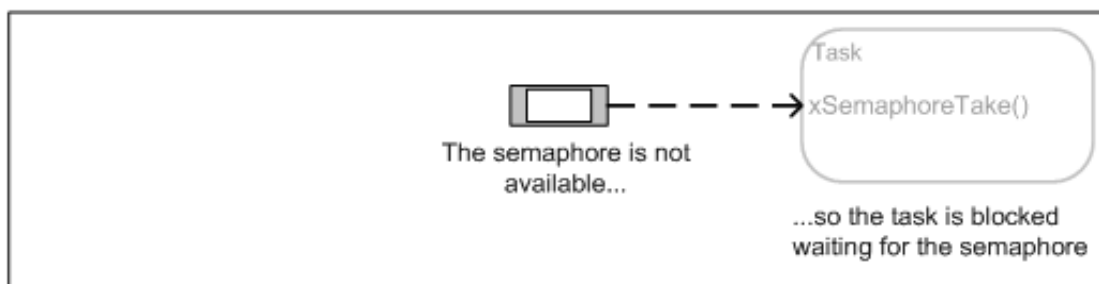


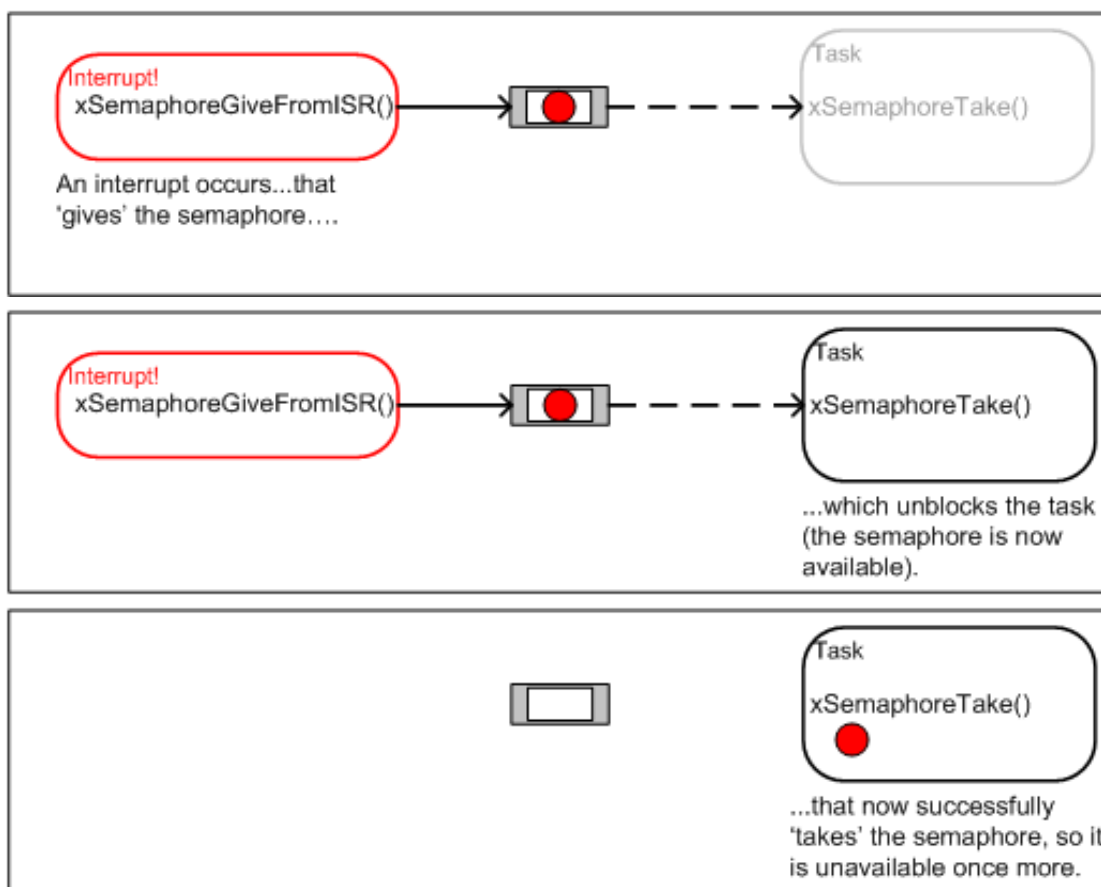
## 4.2 Semaphore Usage

In order 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.

In order 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 ready task 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.

Here is an example describing semaphore usage between hardware interrupt and task:





Going back to our contextual example, the following code in red 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();

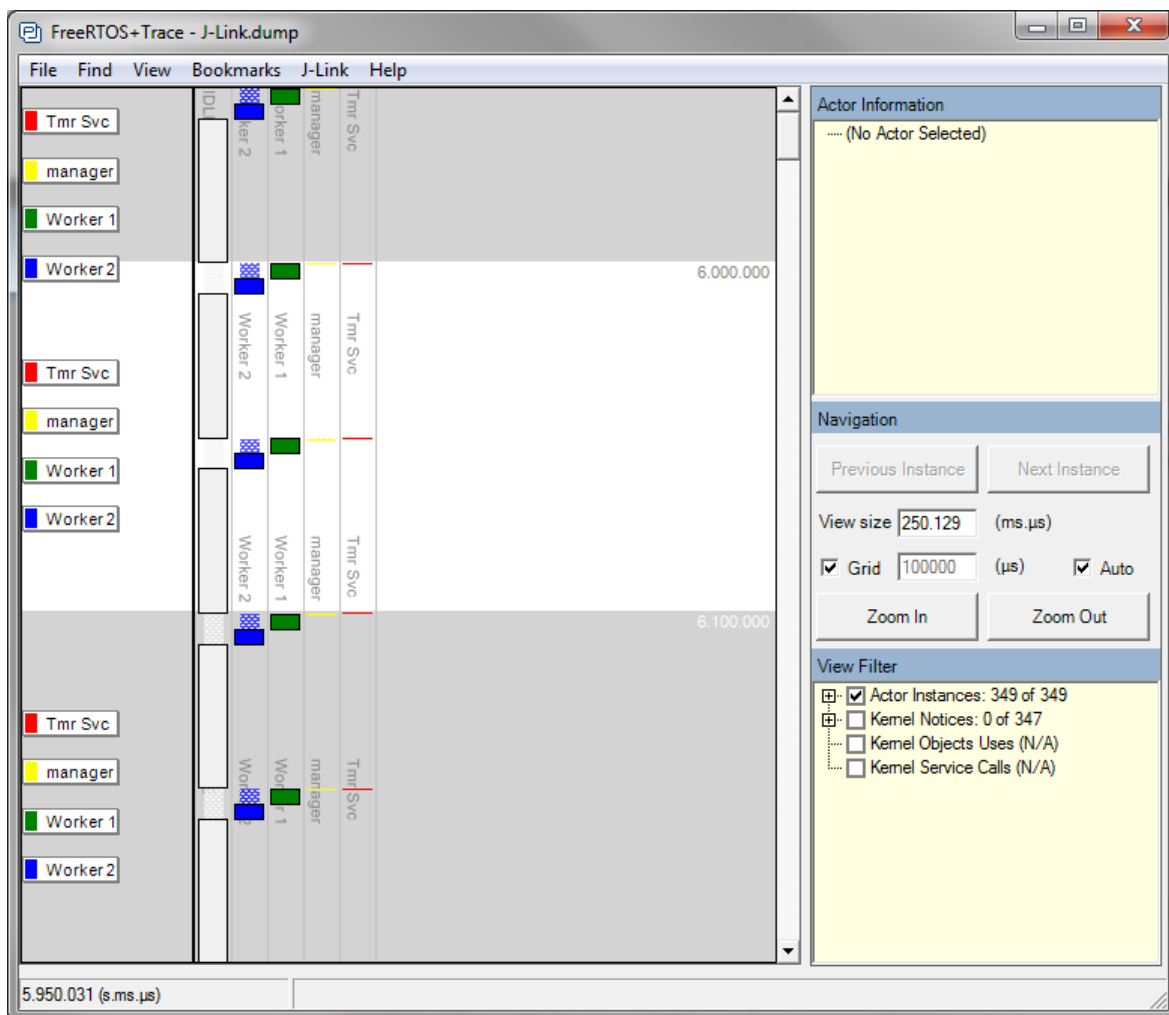
    /*Create 2 Worker tasks. */
    xTaskCreate(worker1_task, "Worker 1", configMINIMAL_STACK_SIZE+1000, NULL, tskIDLE_PRIORITY+1,
    &worker1_id);
    xTaskCreate(worker2_task, "Worker 2", configMINIMAL_STACK_SIZE+1000, NULL, tskIDLE_PRIORITY+2,
    &worker2_id);

    /* 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);
}

```

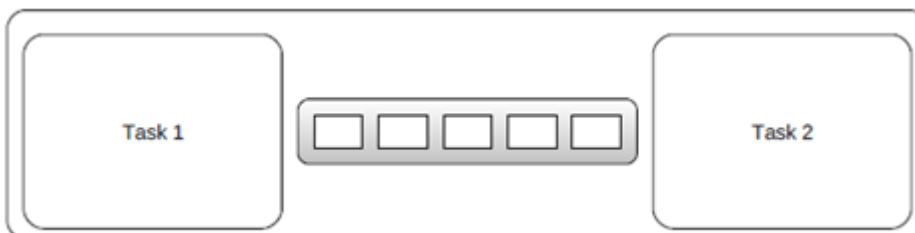


### 4.3 Queue Management

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:

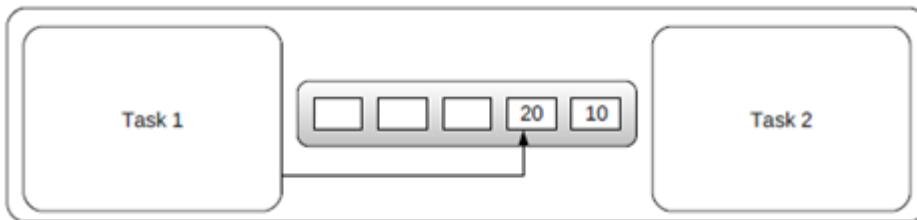
A queue is created to allow task 1 and task 2 to communicate. The queue can hold a maximum of five values. When a queue is created, it does not contain any value so it is empty:



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:



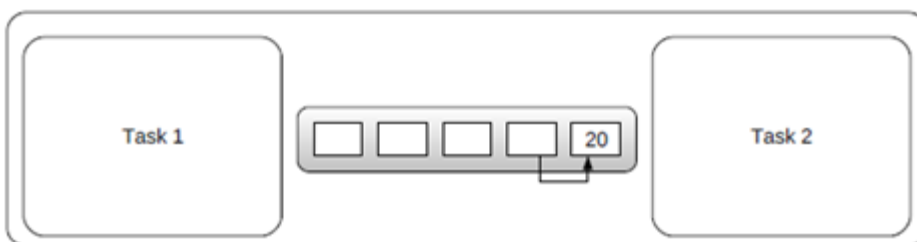
Task 1 sends another value. The queue now contains the previously written value and this newly added value. The previous value remains at the front of the queue while the new one is now at its back. Three spaces are still available:



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



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:



Here is a list of the kernel functions that allows handling of the queue:

- **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

In our contextual example we will illustrate the queue usage by passing CPU load workload simulation information (delay) from manager to Worker task using a message queue (see code highlighted in red).

```

#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);

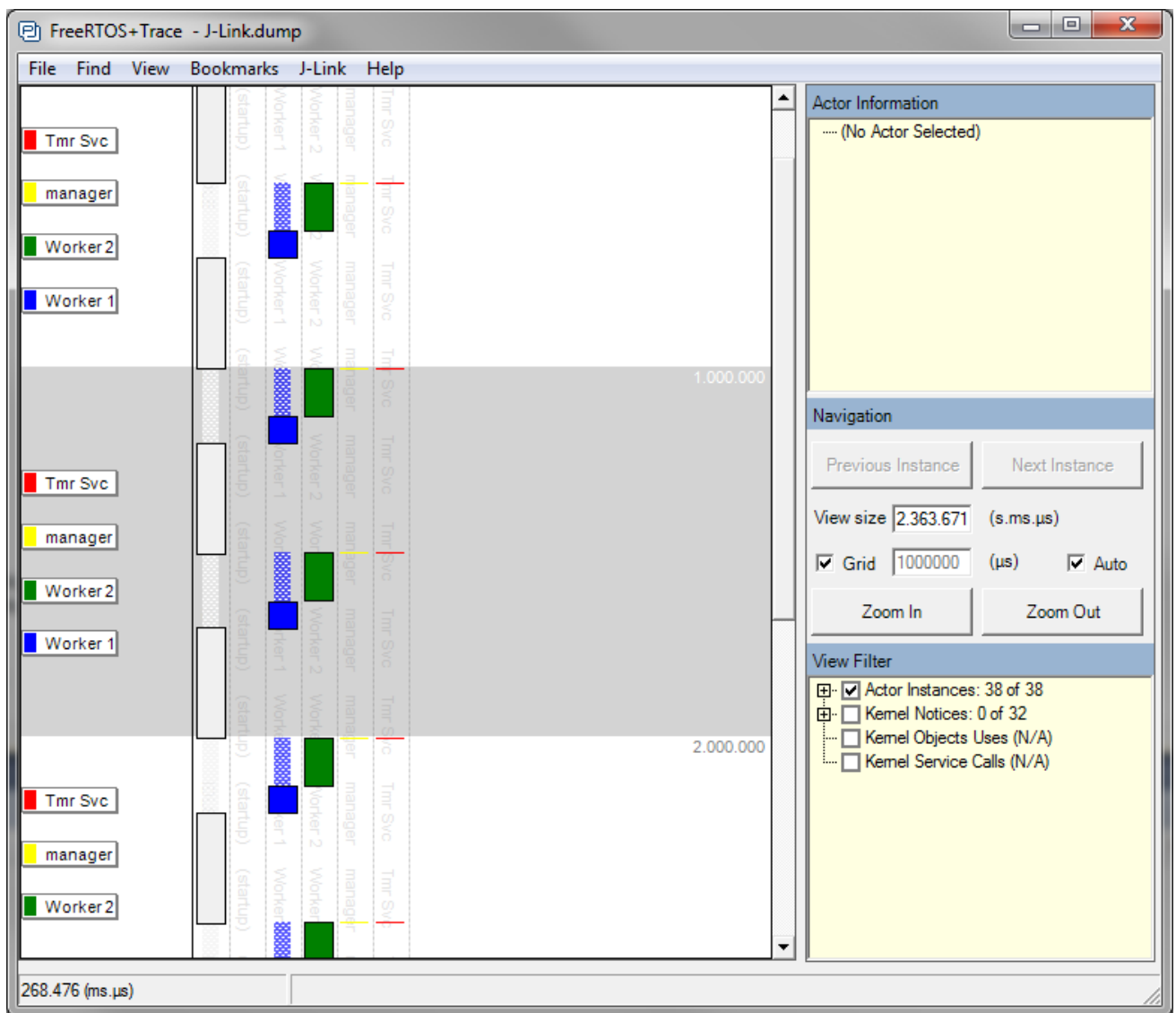
    /* Create one Software Timer */
    Timer_id = xTimerCreate("Timer",500,pdTRUE,0,TimerCallback);
    /* Start Timer.*/
    xTimerStart( Timer_id, 0);

    /* 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);
}

```



## 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 only get executed when there are no tasks of higher priority that are able to run. This makes the idle hook function an ideal place to put the processor into a low power state - providing an automatic power saving whenever there is no processing to be performed (see low power section).

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



It is paramount that the idle hook function does not call any API functions that could cause it to block. Also, 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.



`vApplicationTickHook()` executes from within an ISR so must be very short, not use much stack, and not call any API functions that don't 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.



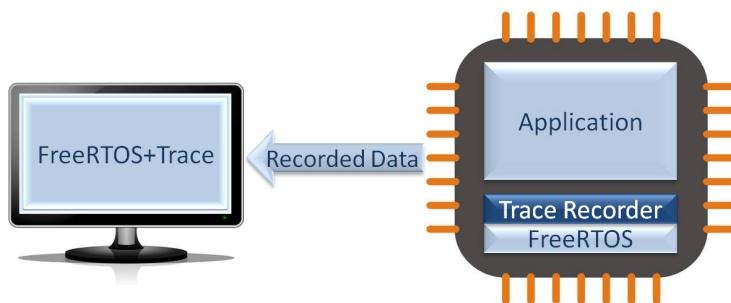
Dynamic memory allocation is not the best to do on MCUs. In the case of FreeRTOS, it is preferable to use dynamic memory allocation only during inits.

## 6 Debugging a FreeRTOS Application

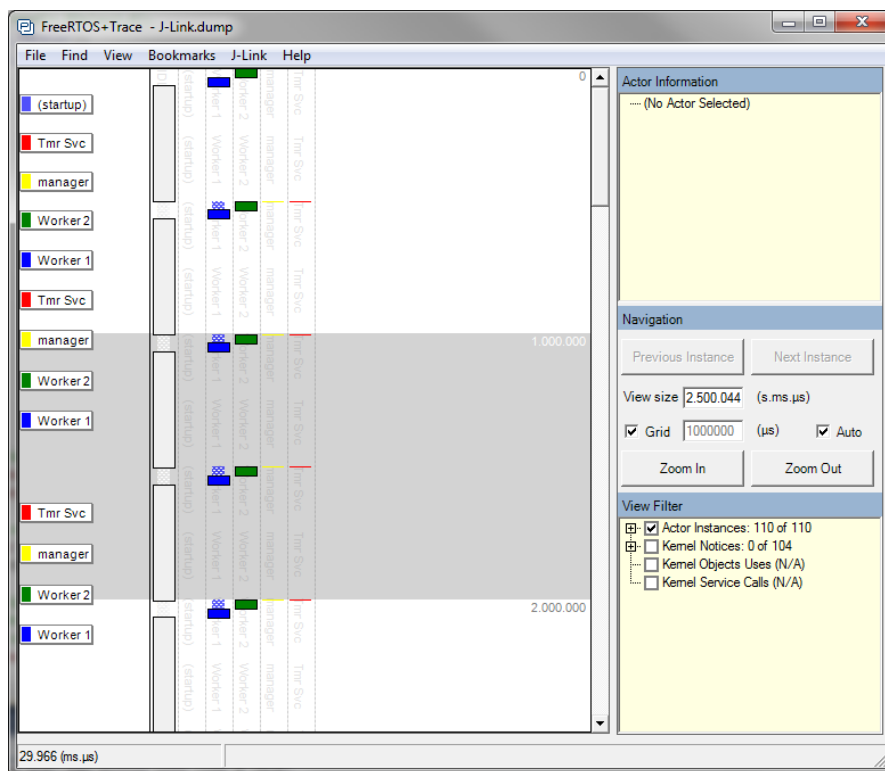
Debugging a real time application is a complex exercise due to multiple task management and kernel object. For this purpose Atmel Studio 6 offers the possibility to download an extension tool called FreeRTOS+Trace.

FreeRTOS+Trace rely on a trace recorder library for FreeRTOS developed by Percepio, in partnership with the FreeRTOS team. This library will allow to records the FreeRTOS kernel events in a dedicated RAM section.

Dedicated PC software will then dump this trace and gives several graphical trace views that explain what happened, showing tasks, interrupts, system calls, and selected application events. This can be used as a lab tool during debug sessions or even in deployed use as a crash recorder if you have storage on the device.

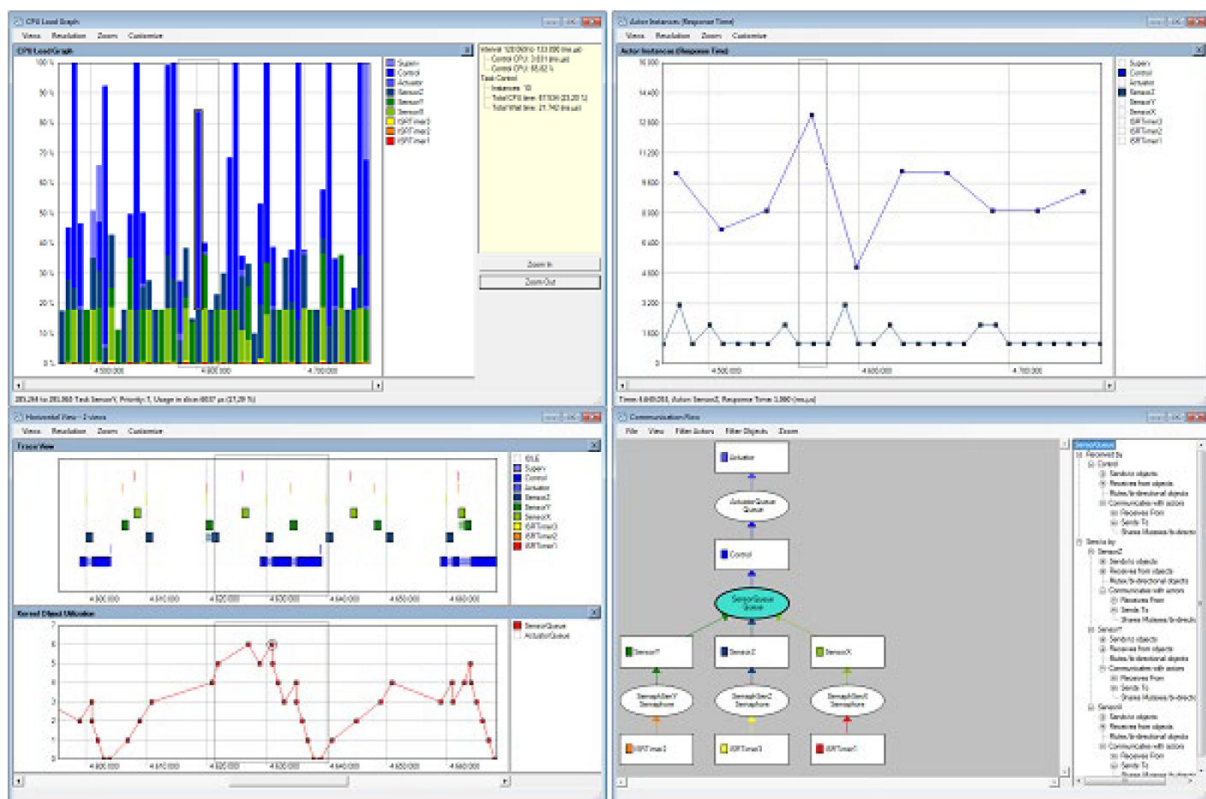


The main trace view shows all recorded events visualized on a common vertical time-line, tasks, and events can be clicked and highlighted for additional information, including timing and dependencies. This gives a detailed understanding when zoomed in on specific intervals and naturally transforms into an overview when zooming out.



In addition to generic task view FreeRTOS+Trace Analyzer allows to analyze:

- CPU Load
- Timing variation
- Communication Flow
- Synchronized view
- Communication flow
- Kernel Object History



## 6.1 FreeRTOS+Trace Integration

FreeRTOS+Trace is available as an Atmel studio 6 extension downloadable via the AS6 extension manager.

The screenshot shows the Atmel Studio 6 Extension Manager interface. The 'Available Downloads' tab is selected, and the 'FreeRTOS+Trace' extension is highlighted. The extension is described as a tool for gaining insight into the runtime world of a FreeRTOS system. It is available for free download. The interface also shows a list of installed extensions and a search bar for available downloads.

**FreeRTOS+Trace**  
Gain an unprecedented insight into the runtime world of your FreeRTOS system using this powerful toolbo...

**Atmel Firmware Tool**  
Standalone tool for upgrading/downgrading firmware on Atmel Tools. Use to downgrade JTAGICE3 from U...

**Created by:** Percepio AB  
**Version:** 2.5.0  
**Downloads:** 3831  
**Rating:** ★★★★★  
[More Information](#)  
[Getting Started](#)  
[Reviews](#)

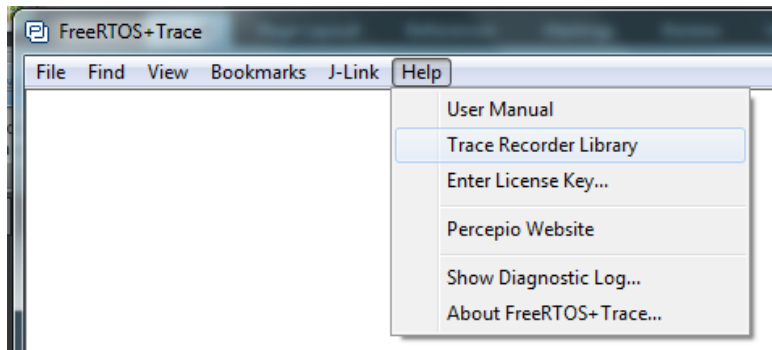
**★★★★★ Mark Sherman**  
4/30/2013 Version: 2.4.1  
This is great!

**★★★★★ sunelf**  
2/7/2013 Version: 2.3.2

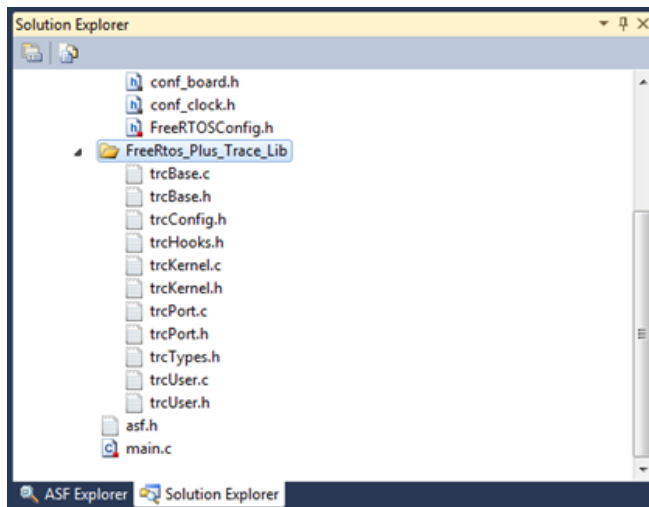
**★★★★★ Tarzan Lin**  
12/19/2012 Version: 2.3.2

As explained in the introduction, this tool is based on a recorder library to add in the project. This library allows enabling trace functionality and allocating a dedicated memory section in the product internal RAM to store traces data for graphical debug.

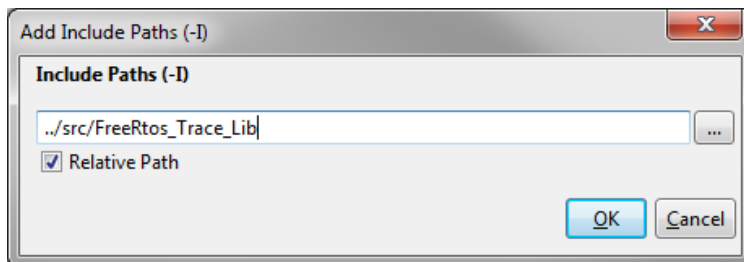
The sources of this library can be found in “” after tool installation or directly by clicking on “**Help → Trace Recorder Library**”.



The inclusion of Trace Recorder Library should be done manually in the Atmel Studio 6 project.



Include the new directory path in the compiler “Directories” properties:



Add the trcHooks library include AT THE END of src/config/FreeRTOSConfig.h file:

```
....

#include "trcHooks.h"

#endif /* FREERTOS_CONFIG_H */
}
```

Add the trace library include at the beginning of your main.c file.

```
#include <asf.h>
#include "trcUser.h"
```

When all, those steps are performed, the Start of Trace recording is done by calling the “uiTraceStart” function from the project main routine.

```
int main (void)
{

    board_init();
    sysclk_init();

    uiTraceStart();

    vTaskStartScheduler();
    while(1);
    // Insert application code here, after the board has been initialized.
}
```

## 6.2 Debug your Application using FreeRTOS+Trace




When project will be executed once, the FreeRTOS debug trace should be available in the SRAM of the SAM4E. We will open **FreeRTOS+Trace** and check that trace data are accessible from the tool.

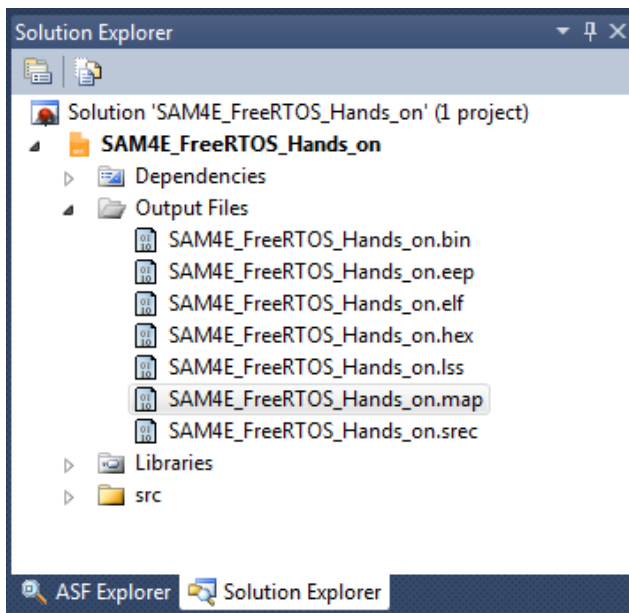
The following process can be followed for debugging an application.



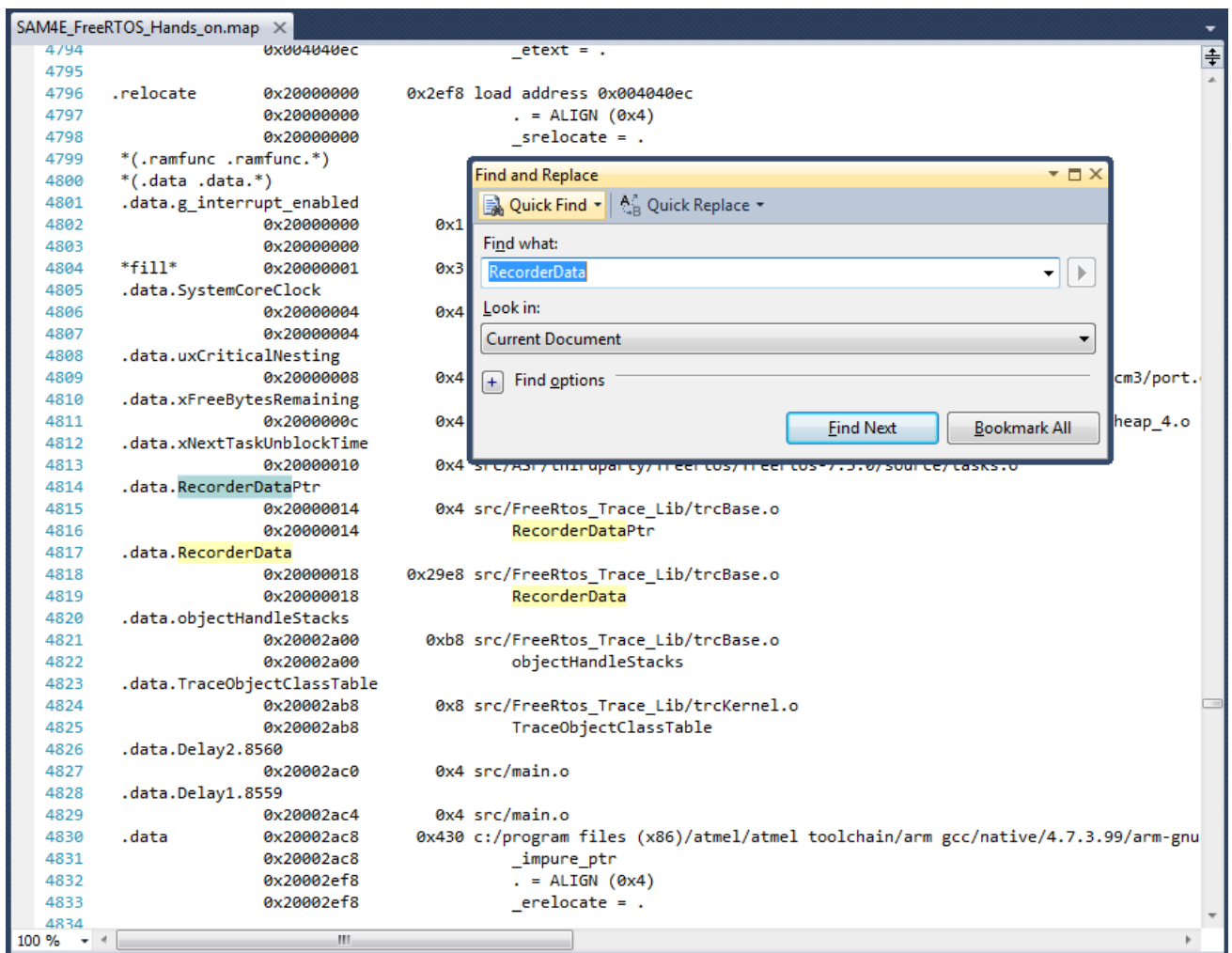
### TO DO

Setup and use FreeRTOS+Trace with your project.

- Connect the board to your computer
- Click on “Build” button: 
- Check the build log in the log output frame to ensure the project built successfully
- Click on “Start Debugging” button  to download and run the program from internal flash of the SAM4E
- Atmel Studio will ask you to select the Debug Tool. Select the SAM-ICE.
- Click on stop debugging button  in order to stop the debug session
- Open the “.map” file of the project. Available in “Output Files” directory.



- Search for “RecorderData” keyword in the .map file (use “Ctrl+F” short key to open find and replace window)



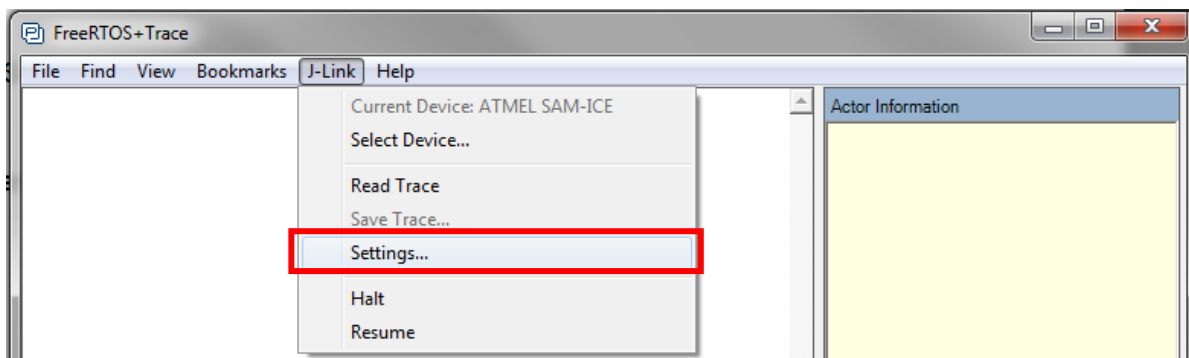
- Retrieve RecorderData section mapping address and size. These data are required by FreeRTOS+Trace to read trace from product memory.

```
.data.RecorderData
0x20000018 0x29e8 src/FreeRtos_Trace_Lib/trcBase.o
0x20000018 RecorderData
```

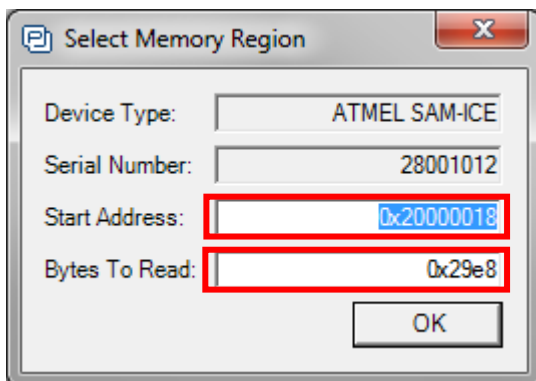


The address and size of the allocated section can be changed according to the selected compiler optimization.

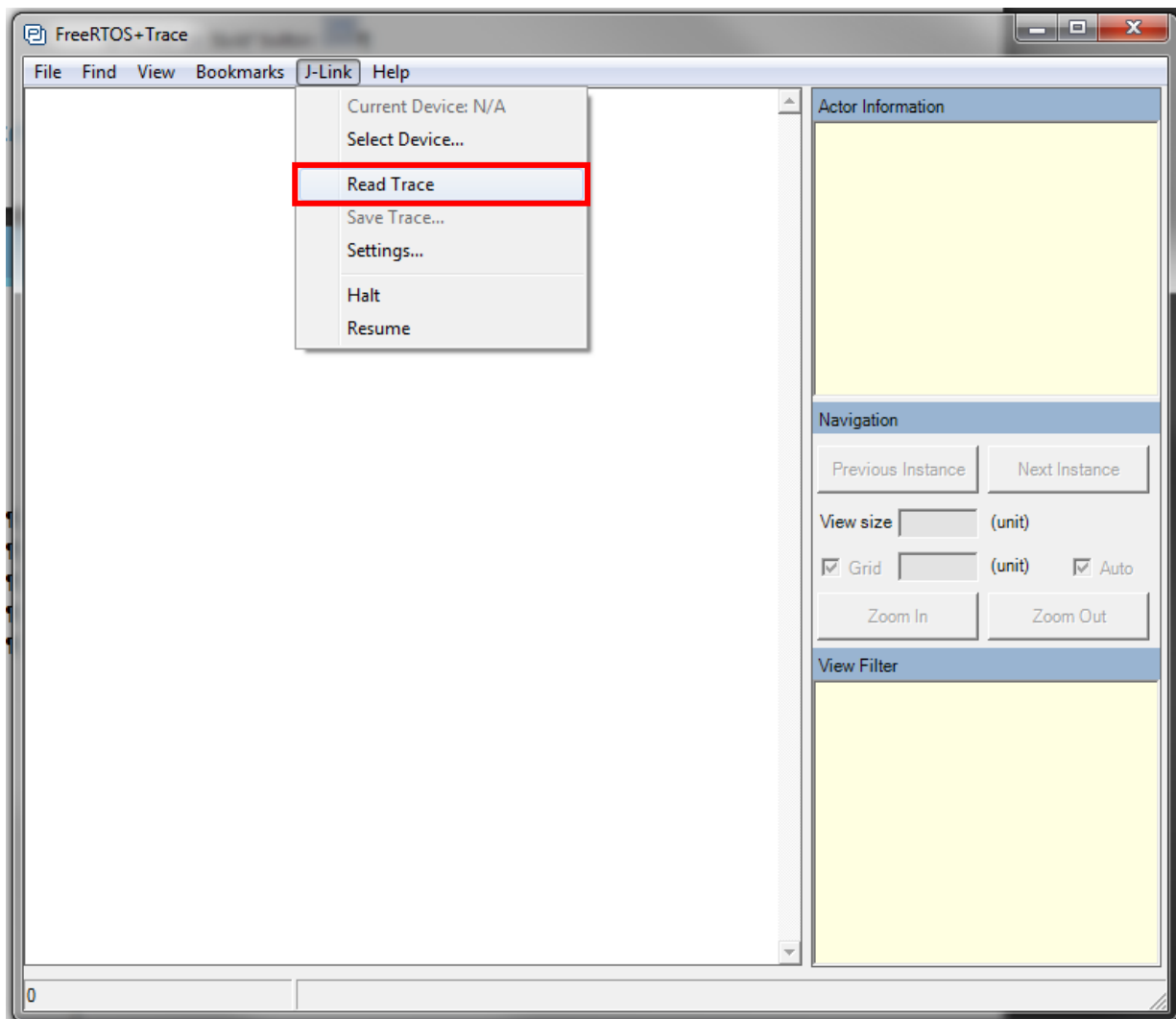
- Under windows, click on “**Start → All Programs → Percepio → FreeRTOS+ Trace → FreeRTOS+Trace**”
- In FreeRTOS+Trace main windows, select “**J-Link → Settings**”



- Configure the start Address and the Bytes to read according to information retrieved from .map file



- Read the project Trace by clicking on “**J-Link → Read Trace**”



## WARNING

If you face some problem when Read/Updating trace from FreeRTOS+Trace follow these steps:

- Close FreeRTOS+Trace
- Use Studio 6.1 to download, run, and then stop debugging
- Restart the FreeRTOS+Trace and then read the trace



## 7 Revision History

Doc Rev.	Date	Comments
42382A	12/2014	Initial document release.



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

© 2014 Atmel Corporation. / Rev.:Atmel-42382A-Getting-Started-with-FreeRTOS-on-Atmel-SAM-Flash-MCUs-ApplicationNote\_122014.

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo, Cortex®, and others are the registered trademarks or trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

**DISCLAIMER:** The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

**SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER:** Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications") without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.