

### Introduction to Basic RTOS Features using SAM4L-EK FreeRTOS Port

AN-4590

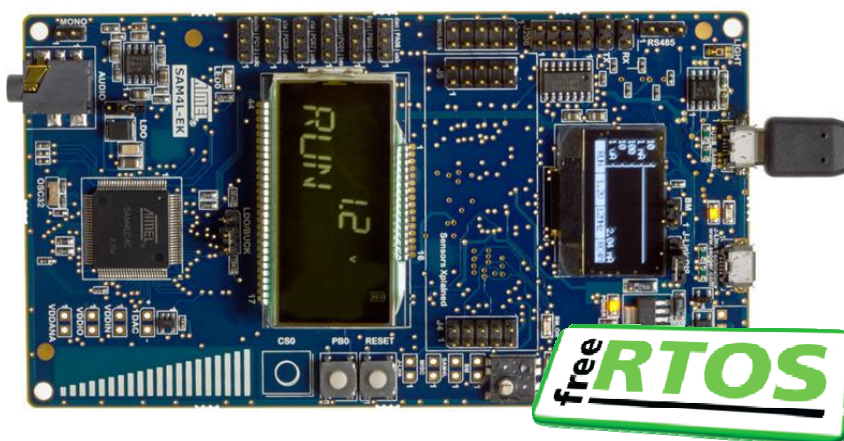
#### Prerequisites

- **Hardware Prerequisites**
  - Atmel® SAM4L-EK Evaluation Kit
- **Software Prerequisites**
  - Atmel Studio 6.1 update 2.0 (build 2730) or higher
  - Atmel Software Frameworks 3.11.0 or higher
  - Latest J-Link / SAM-ICE™ Software & Documentation Pack
  - Percepio FreeRTOS+Trace Analyzer 2.4.0 or higher
- **Estimated Completion Time:** 90 min

#### Introduction

The goal of this Hands-On is to:

- Explain how to Create and configure a FreeRTOS™ Project under AS 6.1
- Give an overview of FreeRTOS mechanism
- Explain how to use FreeRTOS and its Kernel Object
- Show how to make use of FreeRTOS+Trace for Real time project debugging



## Table of Contents







---

Prerequisites .....	1
Introduction .....	1
Icon Key Identifiers .....	3
1. Training Module Architecture .....	4
1.1 Atmel Studio Extension Delivery Case (.vsix).....	4
1.2 Atmel Training Executable Delivery Case (.exe) .....	4
2. Prerequisites .....	5
2.1 FreeRTOS+Trace .....	5
3. Introduction .....	6
3.1 What is a Real-time Application? .....	6
3.2 Real Time Operating System and Multitasking.....	6
3.3 FreeRTOS Introduction .....	7
3.3.1 The FreeRTOS Kernel .....	7
3.3.2 FreeRTOS Tasks Management Mechanism.....	9
3.3.3 Debugging a FreeRTOS Application.....	11
4. Assignment 1: Create and Configure Your FreeRTOS Project	13
4.1 Project Creation under Atmel Studio 6.1 .....	13
4.2 Project Clock Configuration .....	15
4.3 Add and Configure the FreeRTOS Kernel.....	16
4.4 Add Library for FreeRTOS+Trace.....	20
4.5 Compile and Test Your FreeRTOS Project .....	25
5. Assignment 2: Create and Manage Tasks .....	30
5.1 Structure of a Task.....	30
5.2 Task Creation and Deletion .....	31
5.3 Task Management .....	32
5.4 Priority Settings and Round Robin.....	35
6. Assignment 3: Kernel Object Usage .....	39
6.1 Software Timer Usage .....	39
6.2 Semaphore Usage .....	43
6.3 Queue Management .....	47
7. Conclusion .....	53
8. Revision History .....	54

## Icon Key Identifiers

---

Icons are used to identify different assignment sections and reduce complexity. These icons are:

	<b>INFO</b>	Delivers contextual information about a specific topic.
	<b>TIPS</b>	Highlights useful tips and techniques.
	<b>TO DO</b>	Highlights objectives to be completed.
	<b>RESULT</b>	Highlights the expected result of an assignment step.
	<b>WARNING</b>	Indicates important information.
	<b>EXECUTE</b>	Highlights actions to be executed out of the target when necessary.

## 1. Training Module Architecture

This training material can be retrieved through different Atmel deliveries:

- As an Atmel Studio Extension (.vsix file) usually found on the Atmel Gallery web site (<http://gallery.atmel.com/>) or using the Atmel Studio Extension manager
- As an Atmel Training Executable (.exe file) usually provided during Atmel Training sessions

Depending on the delivery type, the different resources needed by this training material (hands-on documentation, datasheets, application notes, software & tools) will be found in different locations.

### 1.1 Atmel Studio Extension Delivery Case (.vsix)

Once the extension is installed, you can open and create the different projects using “*New Example Project from ASF...*” in Atmel Studio.



#### INFO

The projects installed from an extension are usually under “*Atmel Training > Atmel Corp. Extension Name*”.

There are different projects which can be available depending on the extension:

- **Hands-on Documentation:** contains the documentation as required resources
- **Hands-on Assignment:** contains the initial project that may be required to start
- **Hands-on Solution:** contains the final application, which is a solution for this hands-on



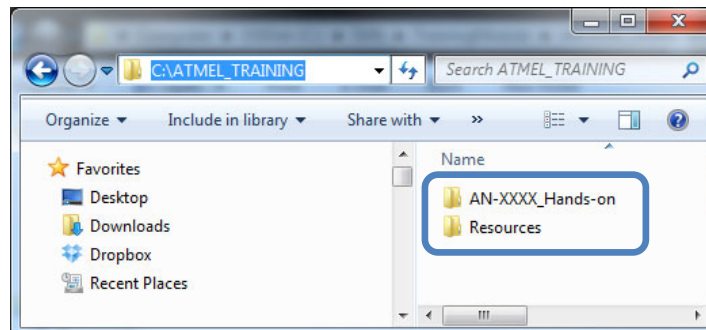
#### INFO

Each time a reference is made to some resources in the following pages, the user must refer to the **Hands-on Documentation** project folder.

### 1.2 Atmel Training Executable Delivery Case (.exe)

Depending on where the executable has been installed, you will find the following architecture, which is composed by two main folders:

- **AN-XXXX\_Hands-on:** contains the initial project that may be required to start and a solution
- **Resources:** contains required resources (datasheets, software & tools...)



#### INFO

Unless a specific location is specified, each time a reference is made to some resources in the following pages, the user must refer to this **Resources** folder.

## 2. Prerequisites



### TIPS

If you got an Atmel Training Executable (.exe), you will also find all the following Software & Tools in the [Resources\Software](#) folder.

### 2.1 FreeRTOS+Trace

Debugging a Real-time application is a complex exercise due to multiple task management and Kernel Object. For this purpose we will work with a tool called FreeRTOS+Trace.



### TO DO

Download and Install FreeRTOS+Trace 2.5.1 or higher.

- Go to <http://percepio.com/tracealyzer/downloads/> and download the latest version
- Execute the [FreeRTOSplusTrace-v2.5.1.exe](#) file and follow the installation wizard



### INFO

If prompted, select the free edition / 30 days evaluation option.



### RESULT

FreeRTOS+Trace is now installed on your computer.

### 3. Introduction

The goal of this hands-on is to illustrate the basic functionality of the FreeRTOS Real Time Operating System and show how to use them in a concrete application. For this purpose, the documentation will go through the following points:

- How to create and configure a FreeRTOS project under Atmel Studio 6
- How to make use of Graphical debugging tool
- How to make use of FreeRTOS basic functionality in an embedded project

During this hands-on we will use a SAM4L-EK and the FreeRTOS kernel port available in Atmel Software Framework (ASF). Before going into more details on FreeRTOS usage and features, it is important to understand what are Real-time applications and Real Time Operating Systems.

#### 3.1 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.

#### 3.2 Real Time Operating System and Multitasking

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

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

### 3.3 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.



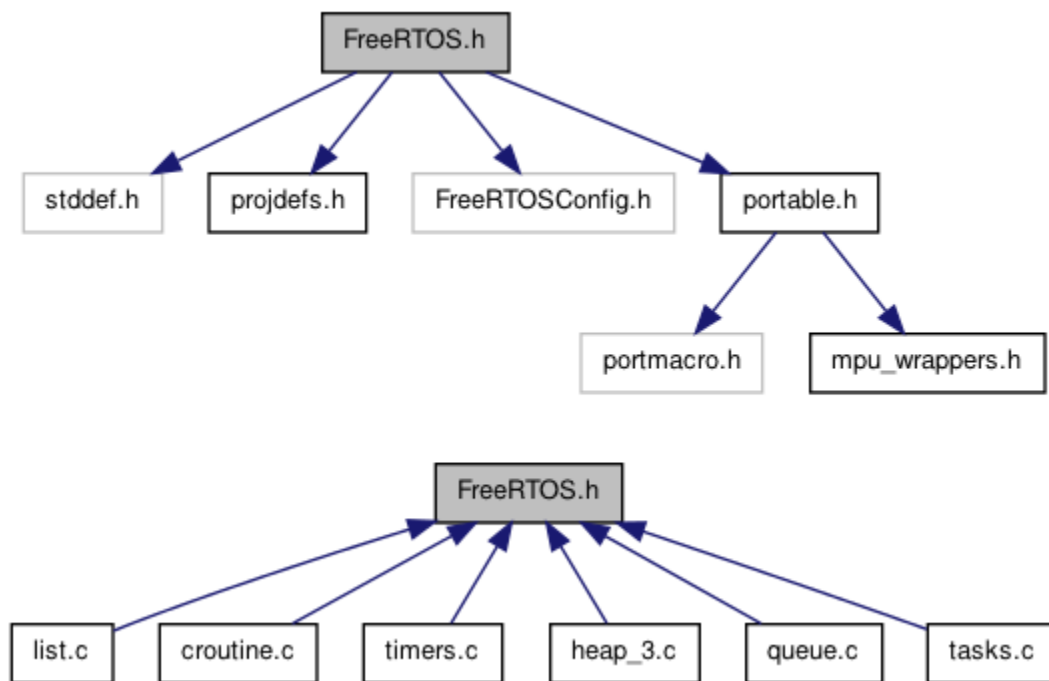
It allows Cortex-M3/M4 microcontroller applications to be organized as a collection of independent tasks to be executed. As most Cortex-M3/M4 microcontrollers have only one core, only one task can be executed at a time.

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 ones.

#### 3.3.1 The FreeRTOS Kernel

FreeRTOS kernel is target independent and is distributed under the Atmel Software Framework as an independent module. 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™.

The FreeRTOS module is made of the following source files.



The Cortex-M3/M4 port includes all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- 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.



#### INFO

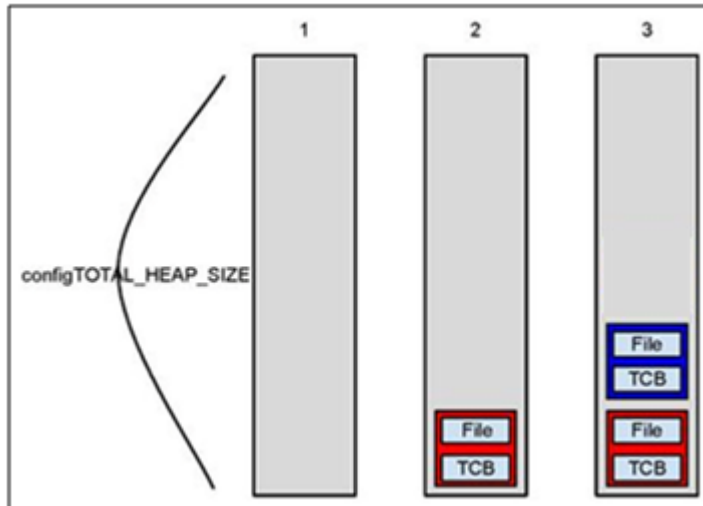
The FreeRTOS kernel is released under GPL with exception, allowing user applications to stay closed source. The BSP part is a mix of GPL with exception license and code provided by the different hardware manufacturers.



### 3.3.2 FreeRTOS Tasks Management Mechanism

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.

The figure below illustrates the memory allocation of tasks in RAM.

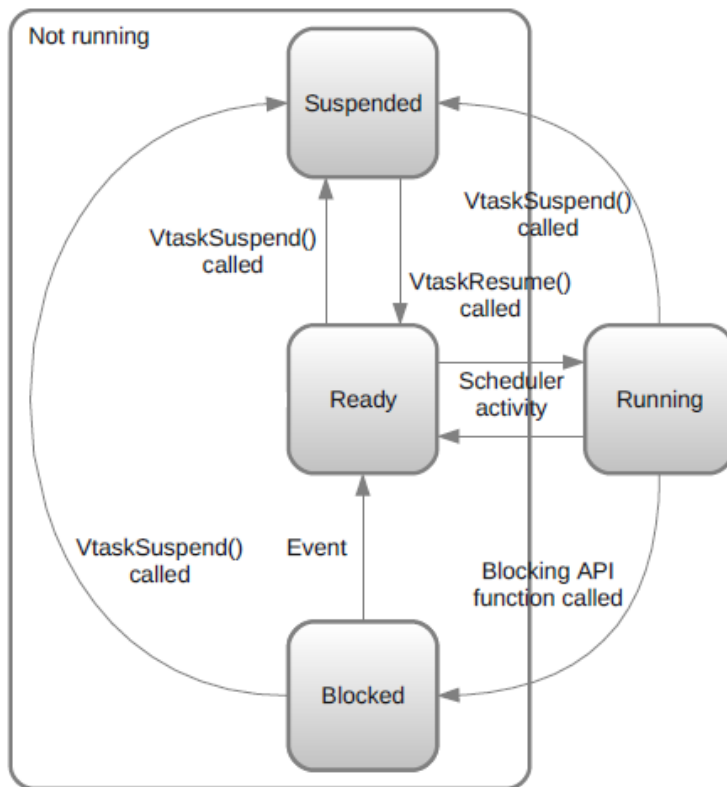


The RTOS kernel allocates RAM each time a task or a kernel object is created. The section allocated to a task or an object is called a stack. The size of this stack is configurable at task creation. The stack contains the “Task File” and the “Task Control Board” (TCB) that allows the kernel to handle 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.

The different states are illustrated in the figure below.

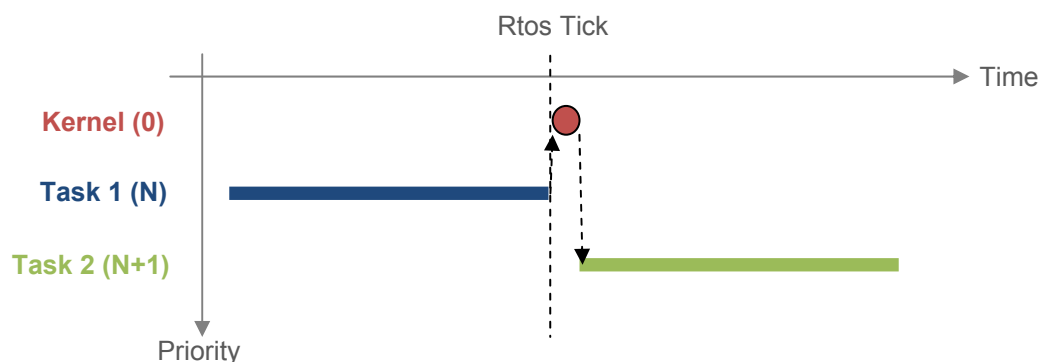


At application level there are two possible states for a task: **Running** and **Not Running**.

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.

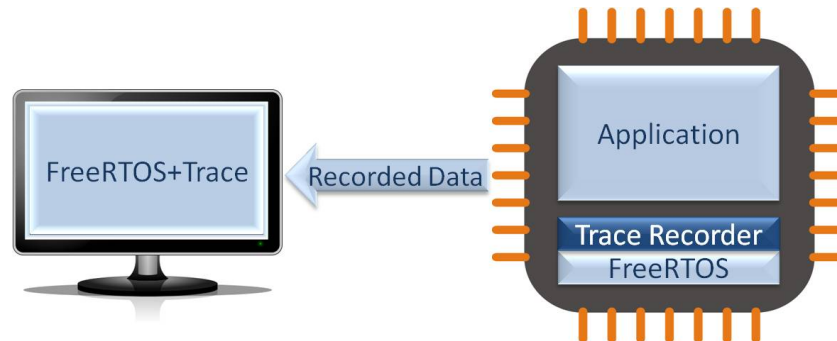


### 3.3.3 Debugging a FreeRTOS Application

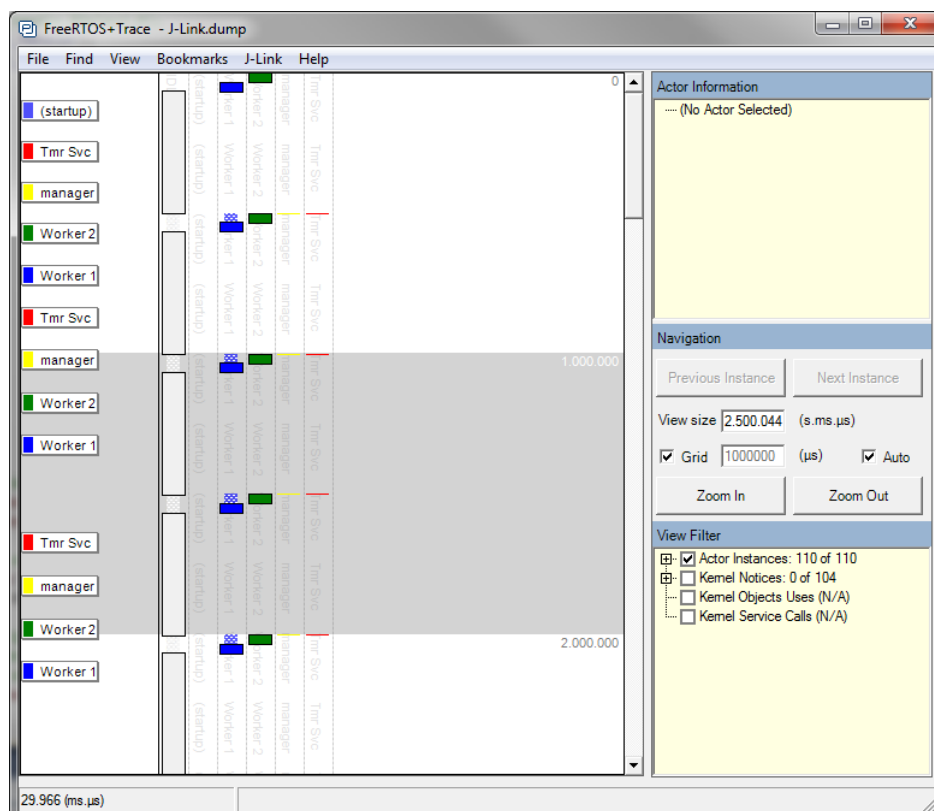
Debugging a Real-time application is a complex exercise due to multiple task management and Kernel Object. For this purpose we will work with a tool called FreeRTOS+Trace.

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.

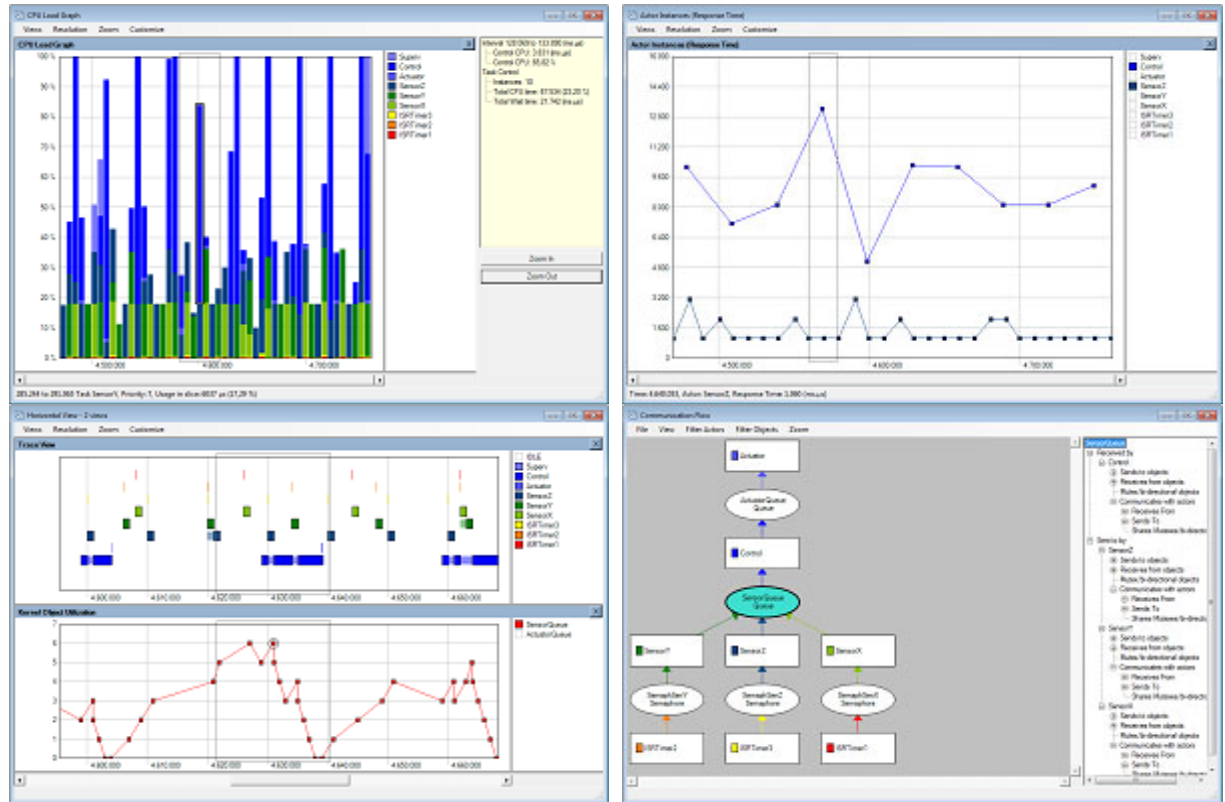


The main trace view shows all recorded events visualized on a common vertical time-line, tasks and events can be clicked and highlighted for additional information, including timing and dependencies. This gives a detailed understanding when zoomed in on specific intervals, and naturally transforms into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



## 4. Assignment 1: Create and Configure Your FreeRTOS Project

Now that we know the basics of FreeRTOS, we can start with our first assignment.

This assignment will guide you through the process of creating a new FreeRTOS project from a pre-built Atmel board template under Atmel Studio 6. It will explain all mandatory steps to be applied each time you create a FreeRTOS project.

This assignment is split into four sections, which explain how to:

- Create the project under Atmel Studio 6.1
- Perform basic configuration of GPIOs and Clocks
- Integrate FreeRTOS Kernel
- Configure FreeRTOS+Trace tool

### 4.1 Project Creation under Atmel Studio 6.1

Atmel Studio embeds different pre-built project templates available when creating a new project from the Atmel Studio home page. These templates are of the following types:

- Atmel board: Lists all the templates linked to Atmel Evaluation kits
- User board: Lists all the templates linked to Custom board
- Atmel Studio solution: Blank solution template

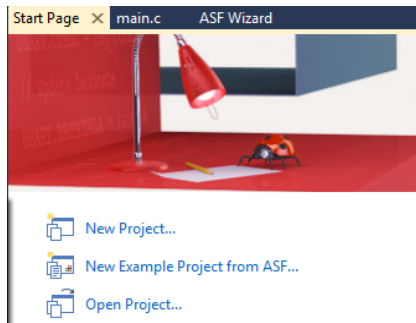
For this hands-on, we will start from the SAM4L-EK template project available in Atmel Board Templates list.



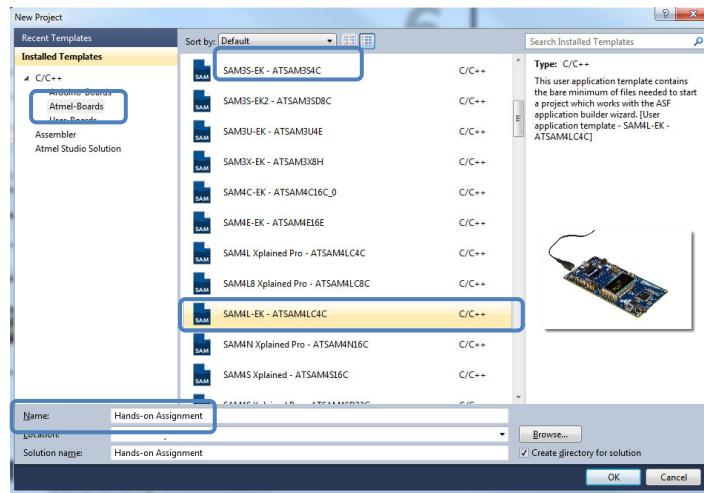
#### TO DO

Create a new project from the Atmel Board template.

- Open Atmel Studio 6.1
- Click on New Project... (or File > New > Project...)



- Select “Atmel Boards > SAM4L-EK – ATSAM4LC4C”
- Rename the Project to “Hands-on Assignment”



- Finally, select the following location to save your new project:

#### Atmel Training Executable Delivery Case

- Save the Hands-on Assignment project to:  
“AN-4590\_SAM4L-EK\_Intro\_FreeRTOS\assignments”  
(folder located in the ATMEL\_TRAINING installation folder)

#### Atmel Studio Extension Delivery Case

- Add the Hands-on Assignment project to the Hands-on Documentation solution
- Click OK



#### **RESULT**

The Solution should appear as in screenshot below in the integrated development environment. You can now configure and customize your project.

## 4.2 Project Clock Configuration

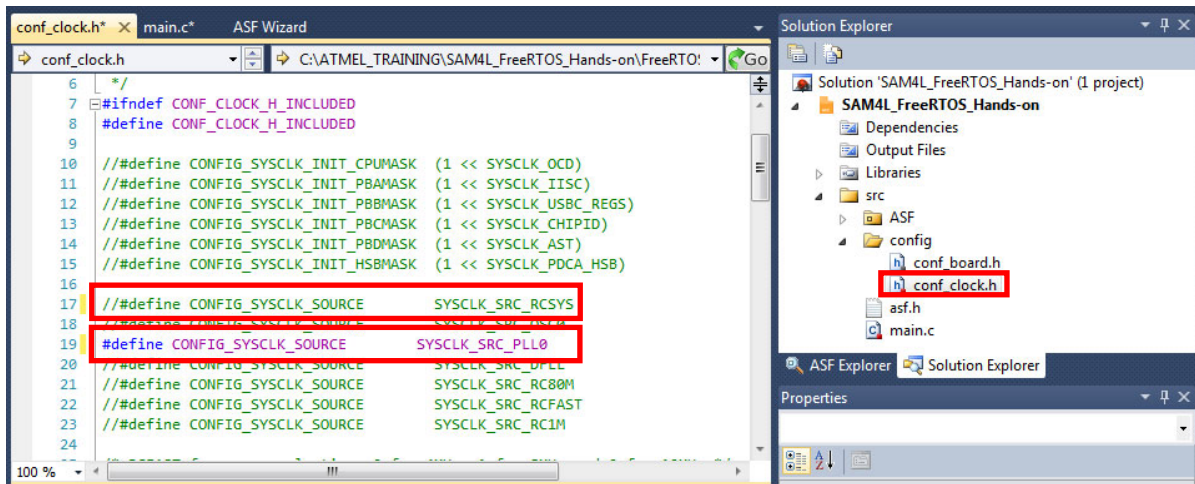
Now that our SAM4L-EK project has been created, we need to perform the basic clock configuration in order to use the SAM4L at maximum speed (48MHz).



### TO DO

Set the project clock configuration to 48MHz.

- Open the `conf_clock.h` available in “src/config” directory from the “Solution Explorer”
- Modify the configuration to use the 48MHz as a clock source:
  - Comment the `#define CONFIG_SYSCLK_SOURCE` `SYSCLK_SRC_RCSYS` line
  - Uncomment the `#define CONFIG_SYSCLK_SOURCE` `SYSCLK_SRC_PLL0` line



- In order to modify the clock speed at project startup, add a call to the `sysclk_init` function at the beginning of the project “main” routine

```
int main (void)
{
    sysclk_init();
    board_init();
    // Insert application code here, after the board has been initialized.
}
```

- Add an infinite loop at the end of the main routine (main routine should not return)

```
int main (void)
{
    sysclk_init();
    board_init();
    while(1);
    // Insert application code here, after the board has been initialized.
}
```



### RESULT

The project is now configured for our application needs and we can continue with FreeRTOS Kernel module Addition.

### 4.3 Add and Configure the FreeRTOS Kernel

We can now start working on the FreeRTOS kernel implementation and configuration.

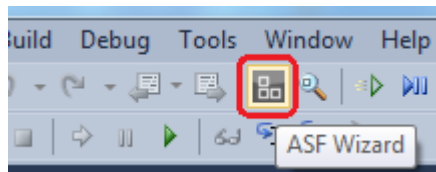
The FreeRTOS kernel is fully integrated in ASF and is a dedicated module that can be added to any standard project using the ASF wizard.



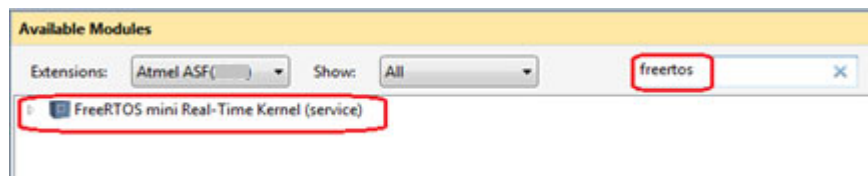
#### TO DO

Add the FreeRTOS kernel.

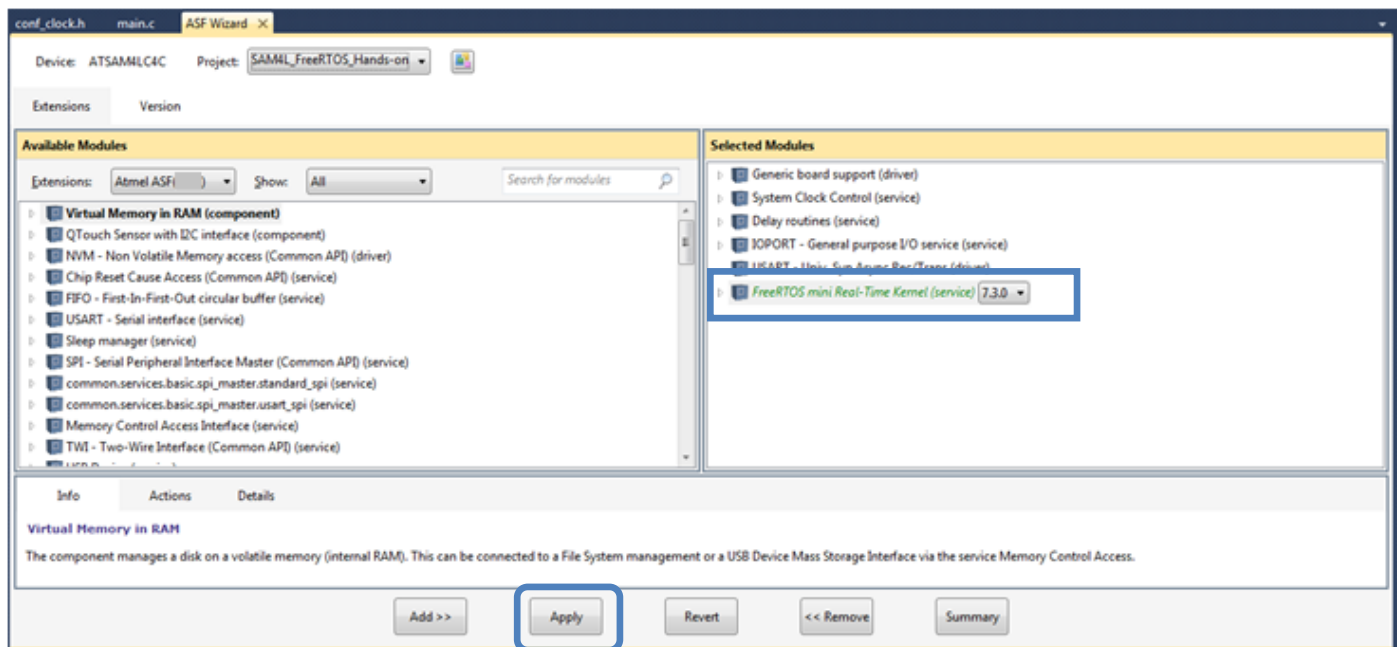
- Click on the ASF wizard icon



- Under *Available Modules*, in the “Search for modules box”, type freertos



- Select “FreeRTOS mini Real-Time Kernel (service)” by clicking on it and click on **Add>>** button. The module will then show up under *Selected Modules*. Click on **Apply** button.

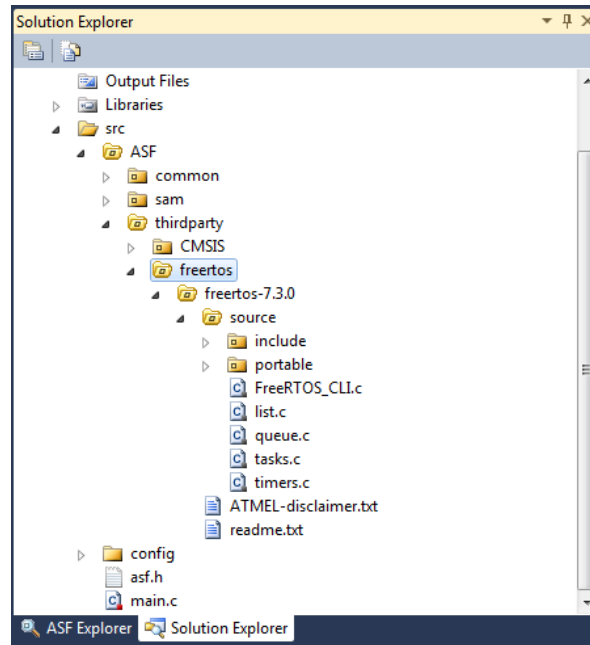


#### INFO

After you click on the **Apply** button, you will be presented with a “Summary of operations” and will also be asked to accept a license agreement.



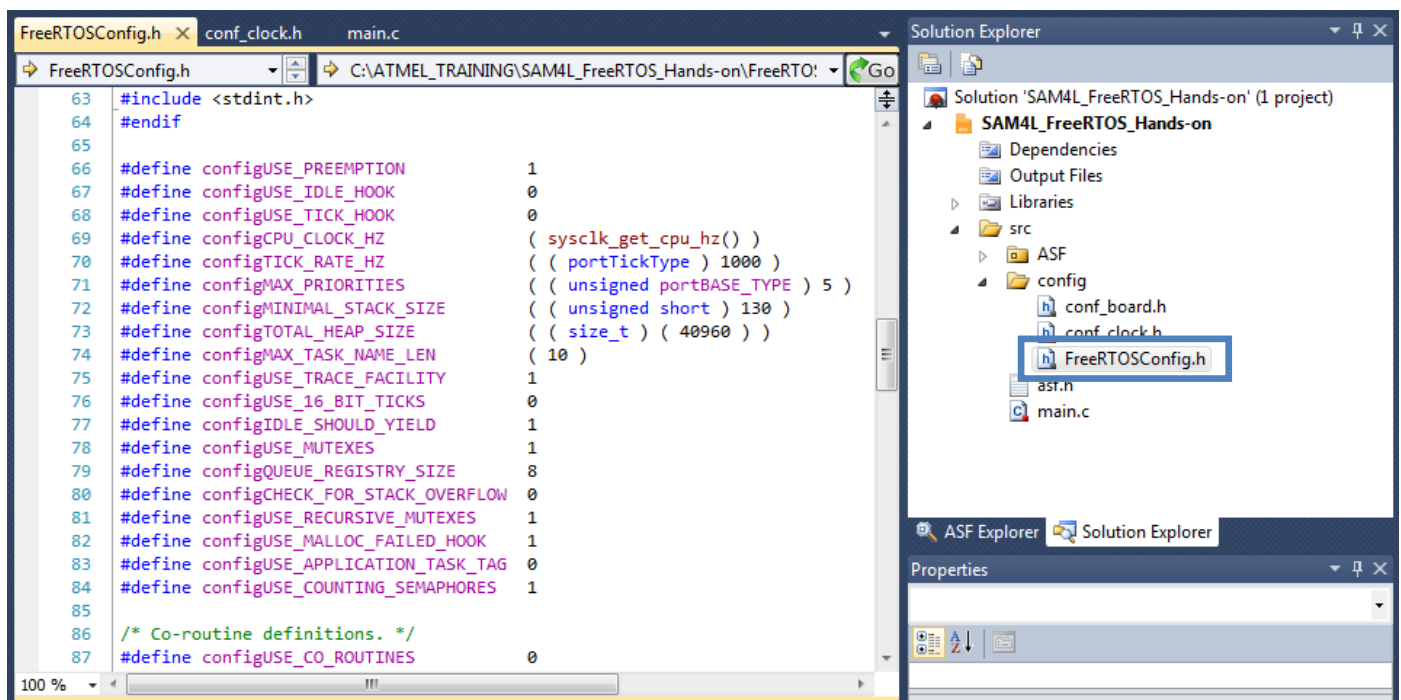
After module addition, the kernel content is available under `src/ASF/thirdparty/freertos/freertos-xxx/` in the project explorer window:



In addition to the “kernel” source files, a configuration file (`FreeRTOSConfig.h`) is added under “`src/config`” directory. This file allows configuring the kernel according to project requirement and also tuning the Kernel integration by setting memory management and excluding unused features from compilation.

Taking time to adapt the kernel to application needs will allow us to reduce the size used by the kernel in memory.

Here is the list of all the basic configuration and customization definition, which can be found in the `FreeRTOSConfig.h` file:



For our application, we will enable the following functionality:

- **USE\_PREEMPTION:**  
Kernel will be used in Preemption mode which means that task with lower priority can be interrupted by task with higher priority.
- **USE\_IDLE\_HOOK:**  
An idle task will be created at scheduler start-up. This task has the lowest priority level possible. It will help us analyze when the CPU has no task to execute.
- **USE\_TRACE\_FACILITY:**  
Enable Kernel Trace function for FreeRTOS+Trace Analyzer.



#### INFO

Additional description can be found on [FreeRTOS.com](http://FreeRTOS.com).



#### TO DO

Configure the FreeRTOS kernel.

- Configure the FreeRTOS kernel by modifying the following definition in the FreeRTOSConfig.h file:

```
#define configUSE_PREEMPTION                1
#define configUSE_IDLE_HOOK                1
#define configUSE_TICK_HOOK                0
#define configCPU_CLOCK_HZ                  ( sysclk_get_cpu_hz() )
#define configTICK_RATE_HZ                  ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES                ( ( unsigned portBASE_TYPE ) 5 )
#define configMINIMAL_STACK_SIZE            ( ( unsigned short ) 130 )
#define configTOTAL_HEAP_SIZE                ( ( size_t ) ( 0x3000 ) )
#define configMAX_TASK_NAME_LEN              ( 10 )
#define configUSE_TRACE_FACILITY            1
#define configUSE_16_BIT_TICKS              0
#define configIDLE_SHOULD_YIELD            1
#define configUSE_MUTEXES                    1
#define configQUEUE_REGISTRY_SIZE            8
#define configCHECK_FOR_STACK_OVERFLOW        0
#define configUSE_RECURSIVE_MUTEXES        0
#define configUSE_MALLOC_FAILED_HOOK        0
#define configUSE_APPLICATION_TASK_TAG      0
#define configUSE_COUNTING_SEMAPHORES        0

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES                0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS                      0
```



#### INFO

Doing so will reduce the kernel memory footprint by removing the unused Kernel features from compilation. Using this configuration, a tick will be generated every 1ms (see configTICK\_RATE\_HZ value).

- Add the following idle hook function in your *main.c* file before main routine declaration

```
#include <asf.h>

void vApplicationIdleHook()
{
    while(1);
}

int main (void)
{
    ...
}
```




#### TIPS

Feel free to directly copy/paste functions in red in Atmel Studio editor.

- Start the FreeRTOS Scheduler using `vTaskStartScheduler` function

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

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

- Click on the “*Build*” button:  to compile your project



#### RESULT

No errors should appear during the compiling process. You have now successfully configured FreeRTOS kernel in your project.

## 4.4 Add Library for FreeRTOS+Trace

The last step of this assignment is to configure the project for using FreeRTOS+Trace.

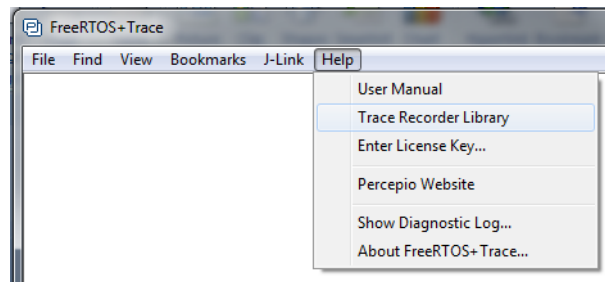
As explained in the introduction, this tool requires a specific library to be added in the project. For the hands-on maintainability purpose, we provide a tested library.

This library will allow the SAM4L to configure the Trace functionality and allocate a dedicated memory section in SRAM to store trace data for graphical debug of the application.



### INFO

Up to date library version maintained by Percepio can be found after FreeRTOS+Trace installation by just clicking on “**Help>Recorder Library**”. Any additional information on the process to add the library to an existing project can be found in Percepio documentation.

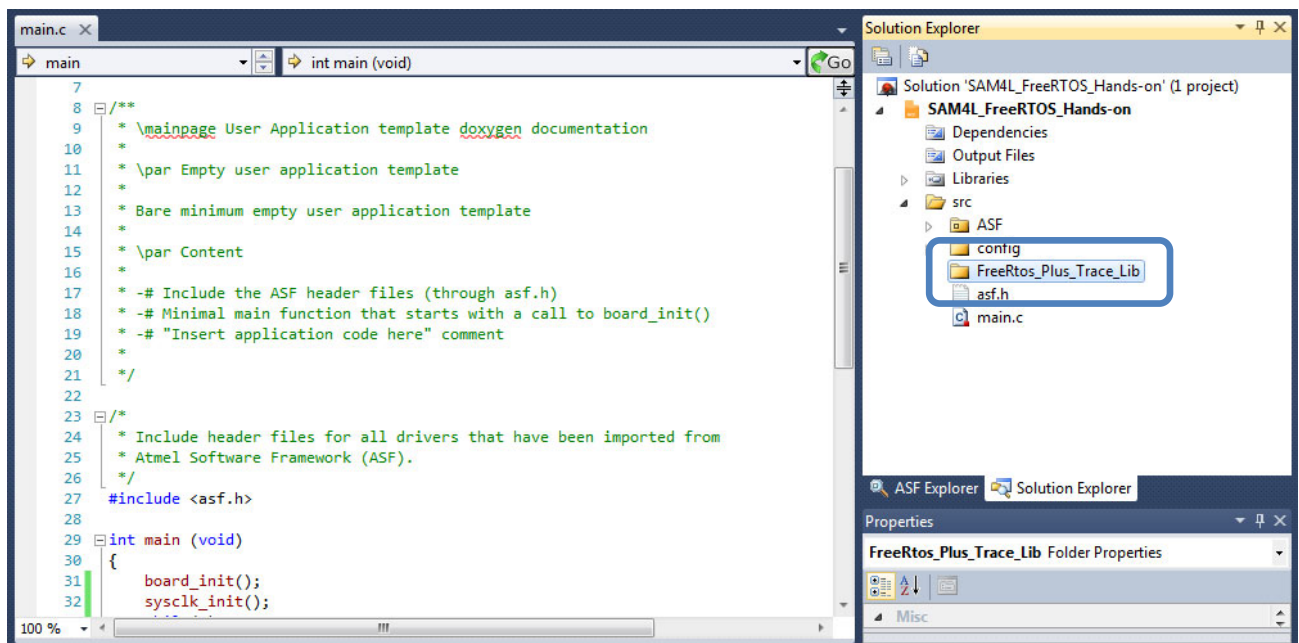
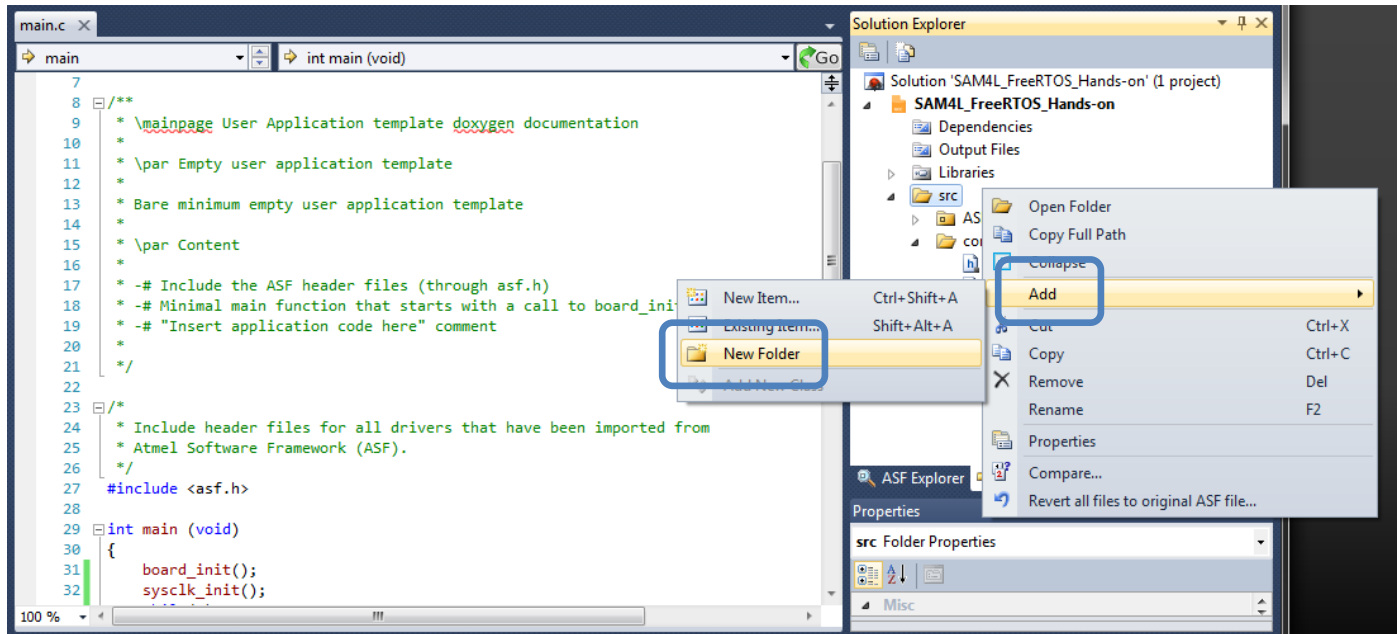




## TO DO

Add FreeRTOS+Trace library.

- Create new folder named “**FreeRtos\_Plus\_Trace\_Lib**” in the src/ directory of your project by right clicking on src and selecting “**Add>New Folder**” under the solution explorer



- Add the whole content of the library by right clicking on the directory **FreeRtos\_Plus\_Trace\_Lib** you just created and selecting “**Add>Existing Item**”

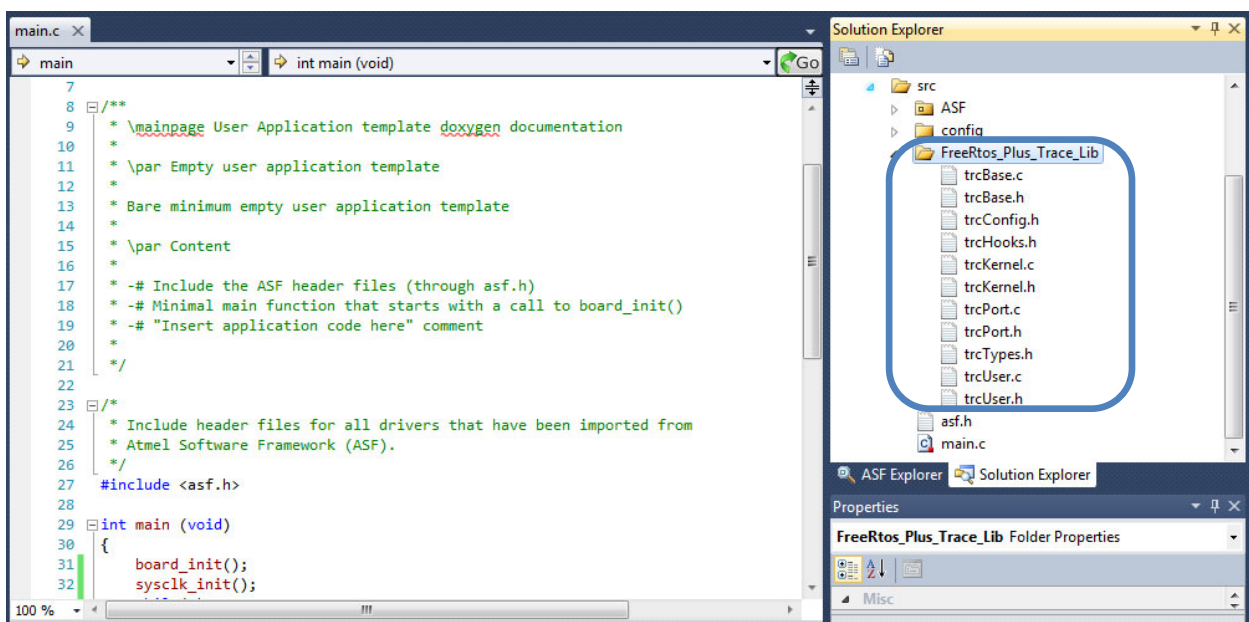
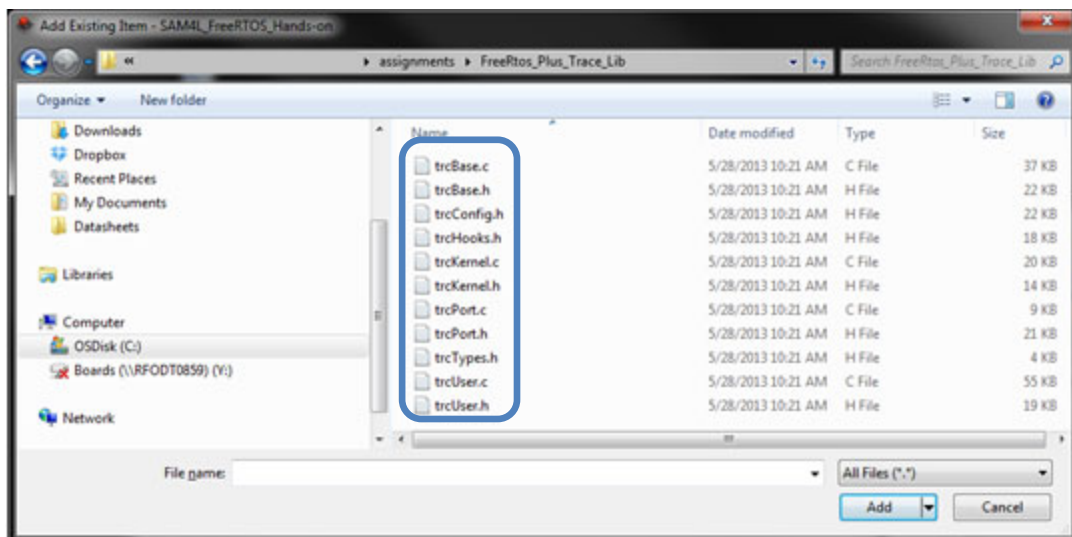
The library files are located in a different location based on the delivery type:

#### Atmel Training Executable Delivery case

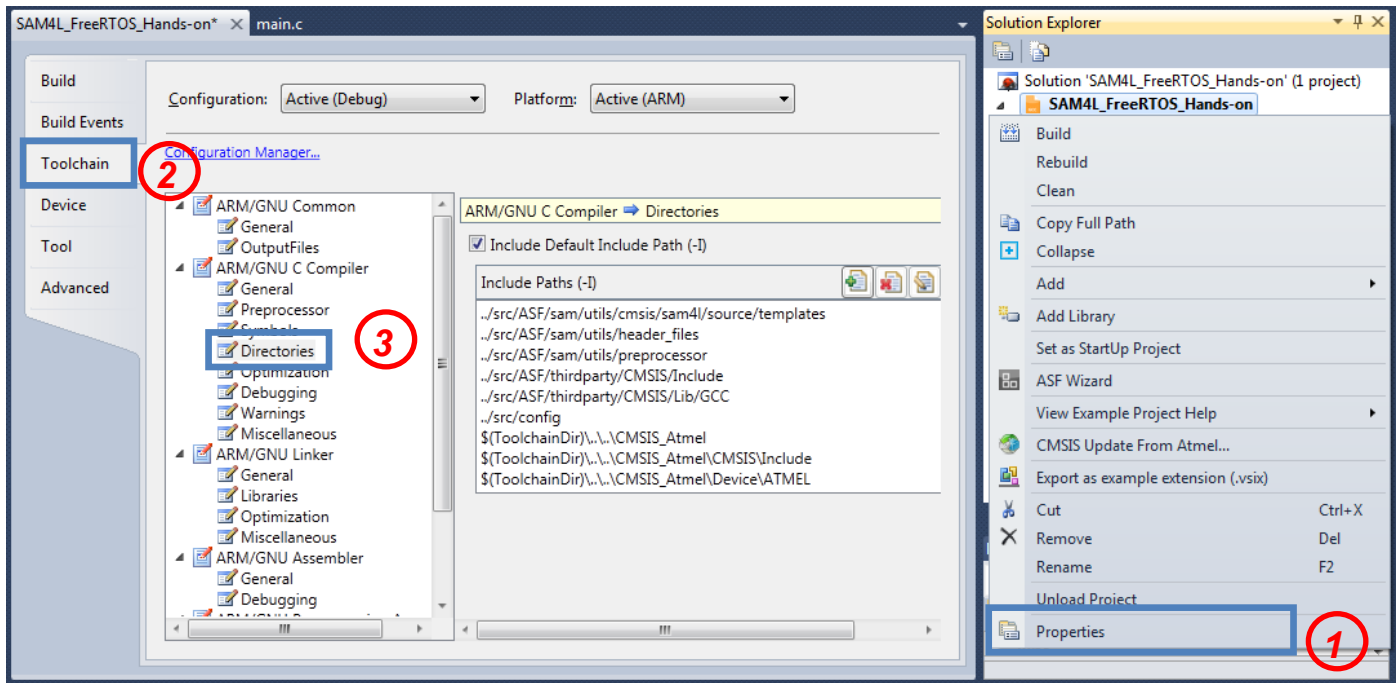
- Find the files in the “AN-4590\_SAM4L-EK\_Intro\_FreeRTOS\assignments\FreeRtos\_Plus\_Trace\_Lib” folder (which is located in the ATMEL\_TRAINING installation folder)


#### Atmel Studio Extension Delivery case

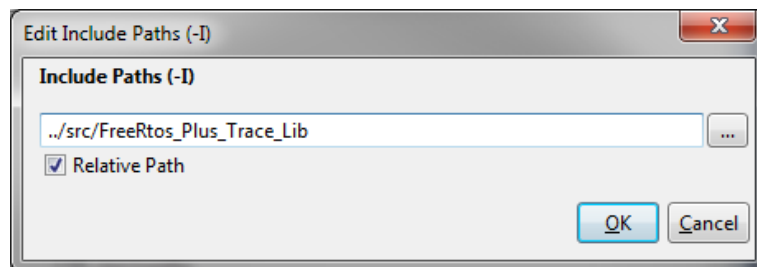
- Find the files in the [Hands-on Documentation](#) project folder



- Include the **FreeRtos\_Plus\_Trace\_Lib** folder under compiler search path
- Access project properties by right clicking on the project Properties under the Solution explorer and then select “**Toolchain>ARM®/GNU C compiler>Directories**”



- Add the following path to the compiler include paths by clicking on Add Item button 



**WARNING** On computer with small size display, you will have to scroll to the right of the frame to see the “Add Item” button.

- 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"
```

- Start the Trace record by calling the “uiTraceStart” function from FreeRTOS+Trace library

```
int main (void)  
{  
  
    board_init();  
    sysclk_init();  
  
    uiTraceStart();  
  
    vTaskStartScheduler();  
    while(1);  
    // Insert application code here, after the board has been initialized.  
}
```



## RESULT

FreeRTOS+Trace is now configured to run with your project.



## 4.5 Compile and Test Your FreeRTOS Project

Now that the project is configured for FreeRTOS usage and debug, we will compile and run it before starting application development.





When the project will be executed once, the FreeRTOS debug Trace should be available in the SRAM of the SAM4L.

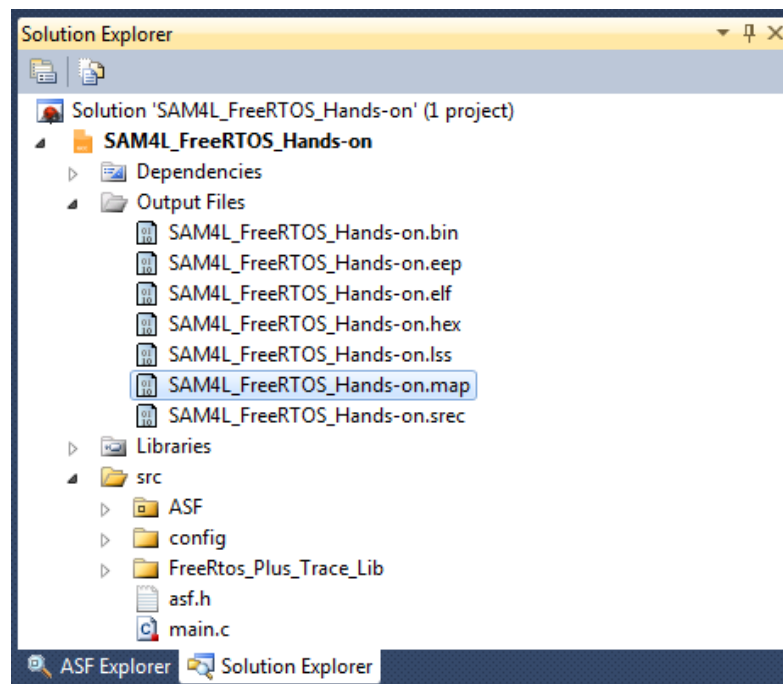
We will open **FreeRTOS+Trace** and check that trace data are accessible from the tool.



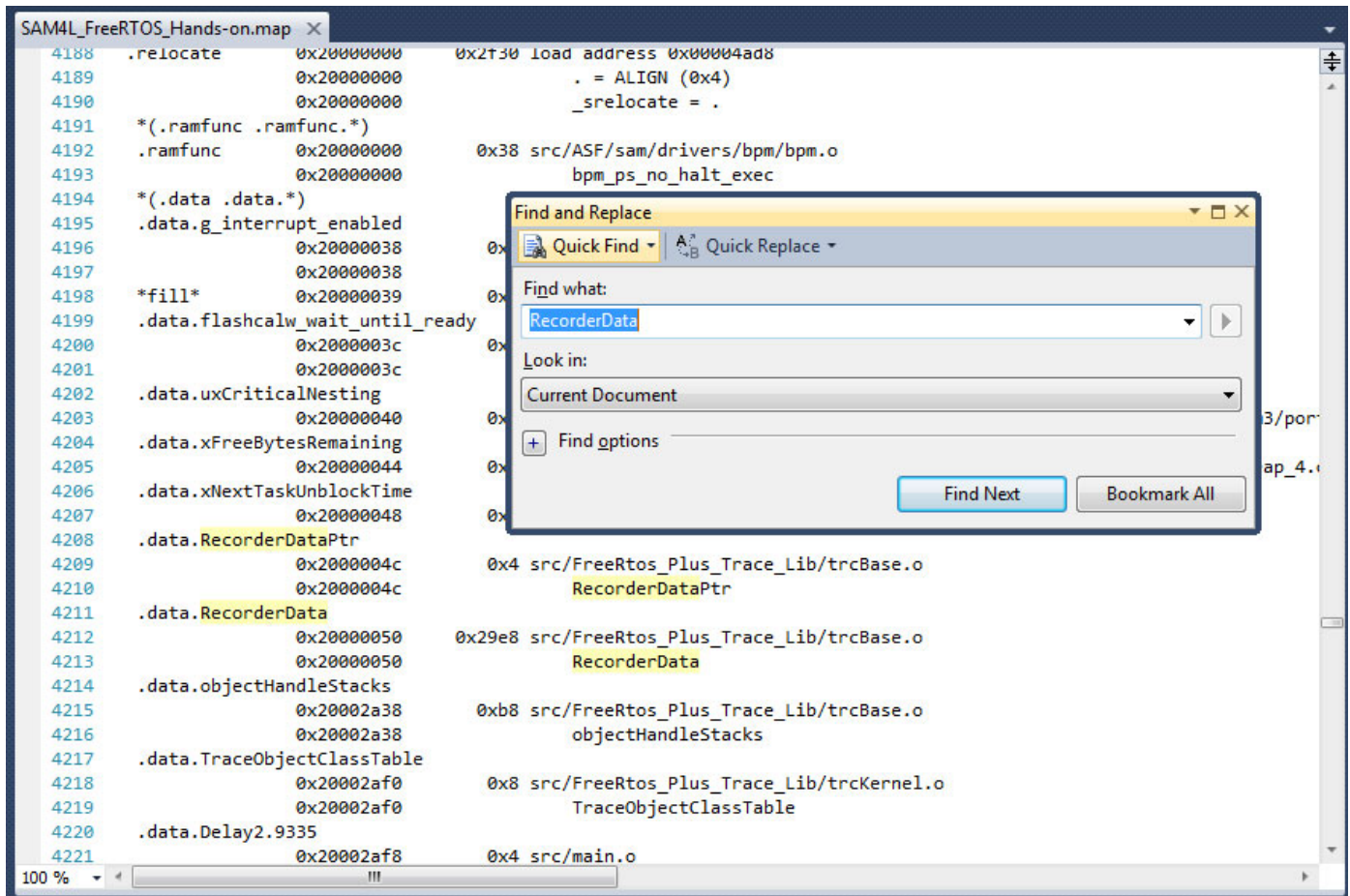
### TO DO

Setup and use FreeRTOS+Trace with your project.

- Connect the board to your computer
- Click on “Build” button:  and 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 SAM4L
- Atmel Studio will ask you to select the Debug Tool. Select J-Link / SWD and click again on 
- 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.

Example:

```
.data.RecorderData
0x20000050 0x29e8 src/FreeRtos_Plus_Trace_Lib/trcBase.o
0x20000050 RecorderData
```



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

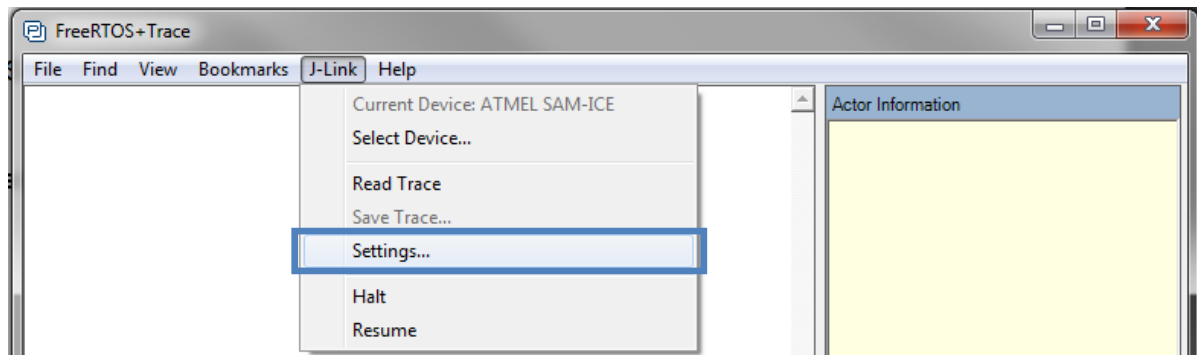
- Under Microsoft Windows, click on “**Start>All Programs>Percepio>FreeRtos+Trace > FreeRtos+Trace**”
- In the Welcome window, select the Free edition / 30days evaluation option



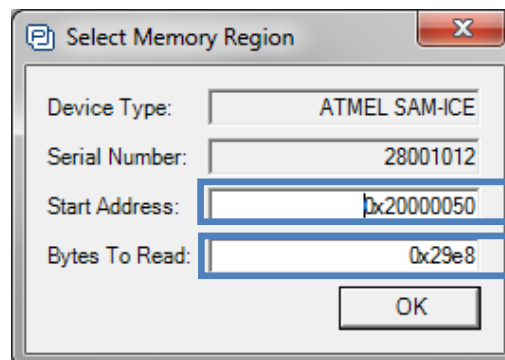
## INFO

Percepio offers a software evaluation version for 30 days. When the evaluation period ends, the "Evaluate" button changes into a "Free Edition" launcher. This hands-on has been developed for the free version.

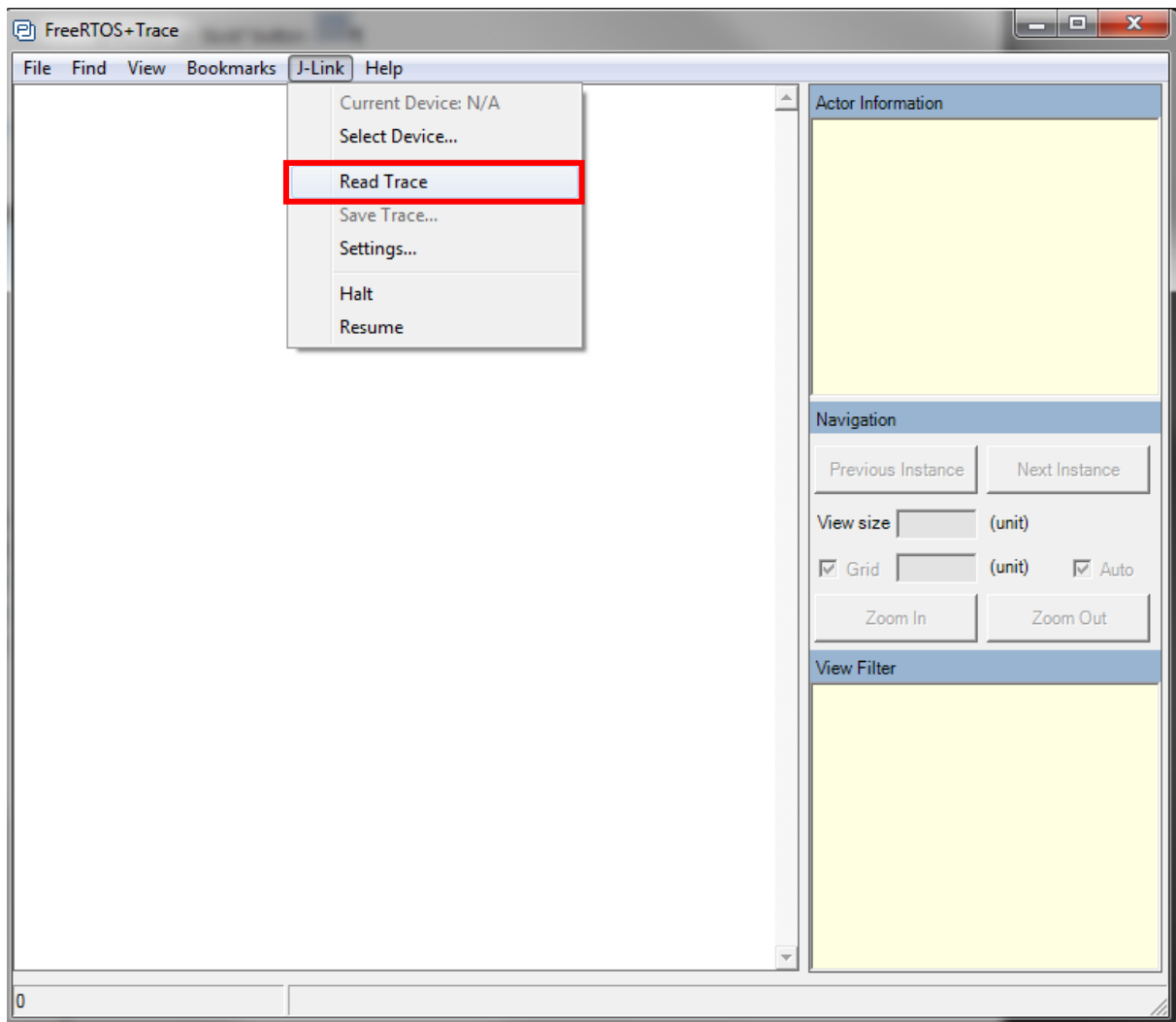
- In FreeRTOS+Trace main windows, Select *J-Link > Settings*



- 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**”

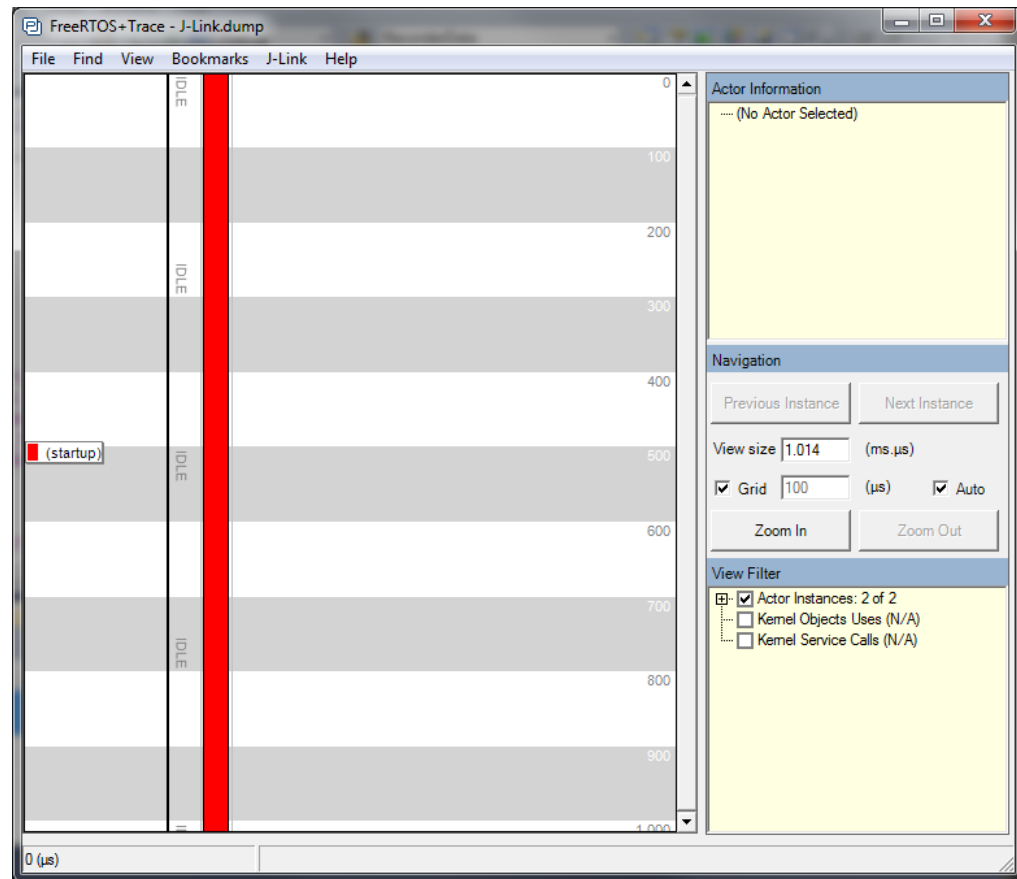


**WARNING** If you face some problem when Read/Updating trace from FreeRTOS+Trace, follow these steps:

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



**RESULT** You should see the following result on “*FreeRTOS+Trace*”.



**INFO** “Startup” corresponds to the time required by the kernel/Scheduler to start.



**RESULT** Congratulations your FreeRTOS project is now configured correctly.

## 5. Assignment 2: Create and Manage Tasks

In this second assignment we will work on the basic task creation, scheduling and handling processes using the FreeRTOS kernel. We will go through the process of:

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

FreeRTOS+Trace will help us to analyze the execution of our code and see the impact of the different kernel function calls and settings.

### 5.1 Structure of a Task

A task is implemented by a function that should never return. They are typically implemented as a continuous loop such as below:

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

As no return is performed, the task should be of void type. A specific structure “parameters” can be used to pass information of any type into the task example:

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

At creation, a handler ID is assigned to each task. This ID will be used as parameter for management function.

```
xTaskHandle task_handle_ID;
```

## 5.2 Task Creation and Deletion

### xTaskCreate:

Tasks are created by calling the xTaskCreate() function provided by the kernel (see task.h, task.c files).

This function has the following Prototype:

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

xTaskCreate function takes the following list of 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 to handle the task. If the task does not have to be handled in the future, this can be NULL

### vTaskDelete:

A task is deleted by using vTaskDelete function. This function has the following Prototype:

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

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.

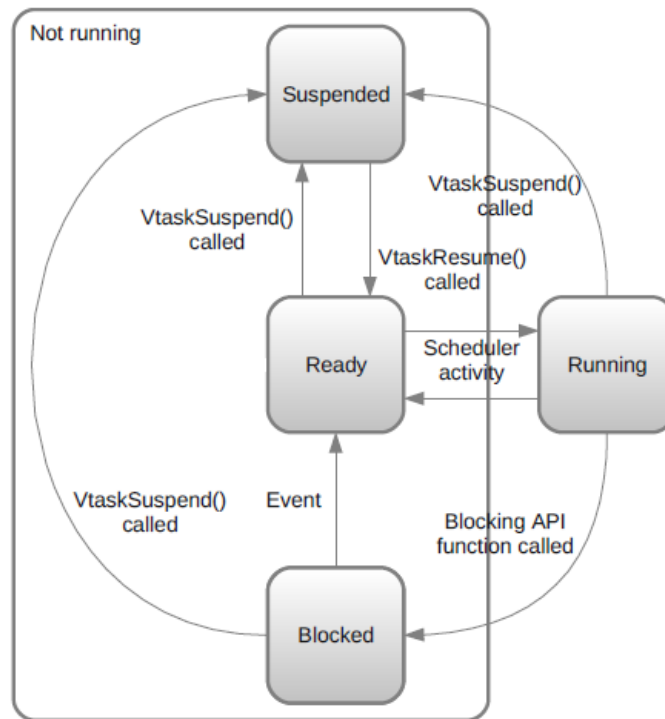


#### TIPS

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

### 5.3 Task Management

FreeRTOS 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:

```
/* Delay a task for a set number of tick */  
void vTaskDelay( portTickType xTicksToDelay );  
  
/* Delay a task for a set number of tick */  
void vTaskDelayUntil( portTickType * const pxPreviousWakeTime,  
portTickType xTimeIncrement );  
  
/* Set task priority */  
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE  
uxNewPriority );  
/* Retrieve Task priority setting */  
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );  
  
/* Suspend a Task */  
void vTaskSuspend( xTaskHandle pxTaskToSuspend );  
  
/* Resume a Task */  
void vTaskResume( xTaskHandle pxTaskToResume );  
  
/* Retrieve the current status of a Task */  
eTaskState eTaskStateGet( xTaskHandle pxTask );  
  
/* Delete a Task */  
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Most of these functions take as parameter a `pxCreatedTask` ID, which is given when the Task was created.



The first step of this assignment is to create two tasks that simulate a CPU workload using a Delay loop (Delay = 100000) then suspend themselves.



## TO DO Create two Tasks.

- In main.c file, define two global task Handlers “worker1\_id” and “worker2\_id”




```
xTaskHandle worker1_id;  
xTaskHandle worker2_id;
```

- Implement the following tasks function above the main routine

```
static void worker1_task(void *pvParameters)  
{  
    static uint32_t idelay, 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 , 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);  
}
```

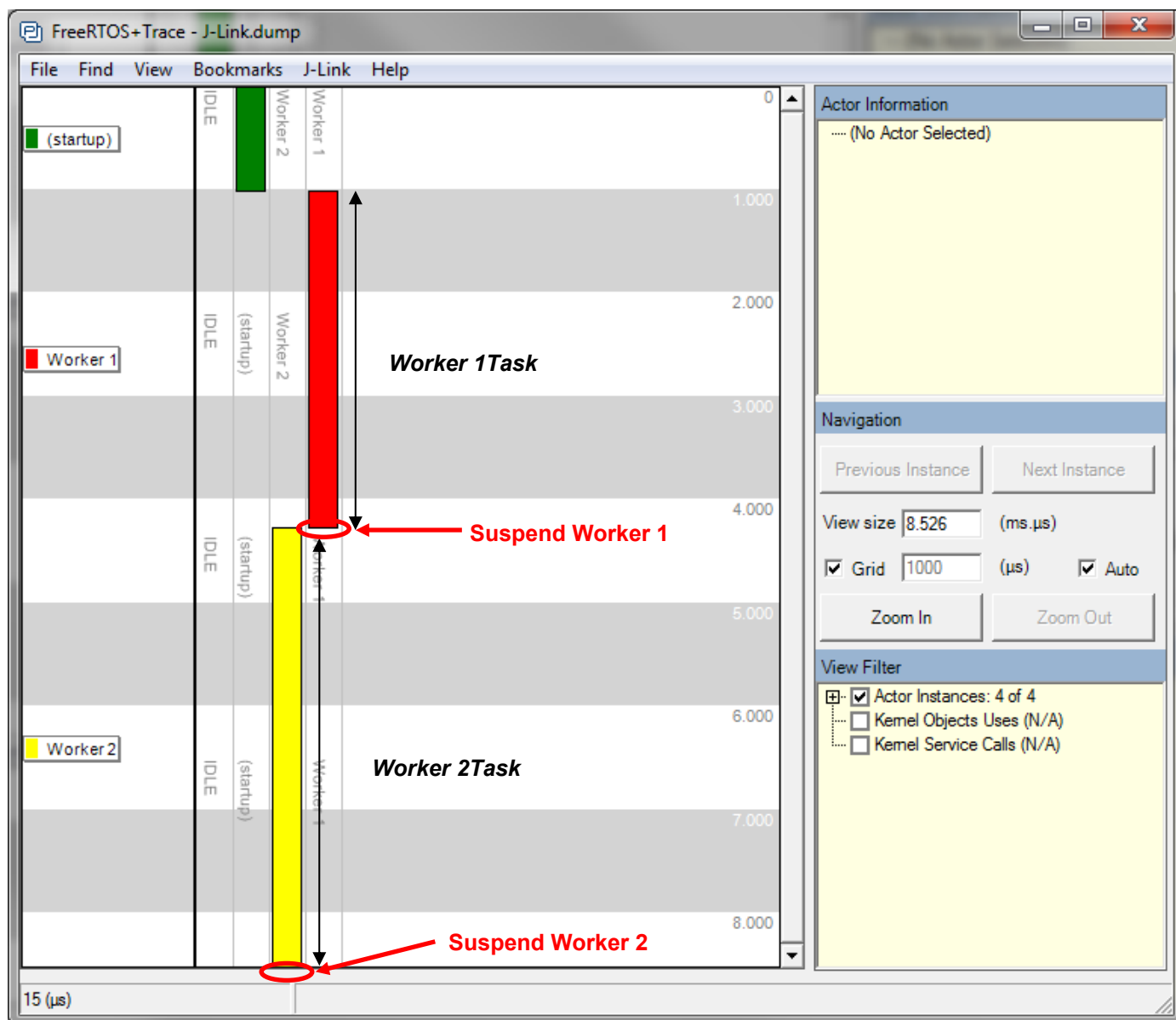
- In the main routine add the following code lines in order to create two worker tasks before starting the scheduler

```
/* Create Worker 1 task */  
xTaskCreate(worker1_task, "Worker 1", configMINIMAL_STACK_SIZE+100, NULL, 2, &  
worker1_id);  
/* Create Worker 2 task */  
xTaskCreate(worker2_task, "Worker 2", configMINIMAL_STACK_SIZE+100, NULL, 1, &  
worker2_id);
```

- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS + Trace output by clicking on “J-Link>Read Trace”



**RESULT** You should see the following result on “FreeRTOS+Trace”.



We can see on the trace report that each task is scheduled and launched. Each task will load the CPU using a loop for a certain time, then suspend (see following extract from task1 function).

```
Delay = 100000;
...
/* Load the CPU for a iDelay Time. */
for (idelay = 0; idelay < Delay; ++idelay);
/* Suspend the task */
vTaskSuspend(worker1_id);
```



#### INFO

In these Task functions, we decide to use an empty loop to simulate a CPU usage. This has been done for hands-on comprehension purpose. But you can use any MCU IPs or features in a task.

## 5.4 Priority Settings and Round Robin

Now that we know how to create tasks, we can work on their scheduling.

FreeRTOS allows developer to affect different level of priority for each task to execute.

This kernel functionality is called Preemption. The task priority is performed during task creation (xTaskcreate parameter).

See the following example:




```
/* Create Worker 1 task */
xTaskCreate(worker1_task, "Worker 1", configMINIMAL_STACK_SIZE+100, NULL, 2 & worker1_id);
/* Create Worker 2 task */
xTaskCreate(worker2_task, "Worker 2", configMINIMAL_STACK_SIZE+100, NULL, 1 & worker2_id);
```

Using different priority combination will have a different impact on the tasks execution.



#### TO DO

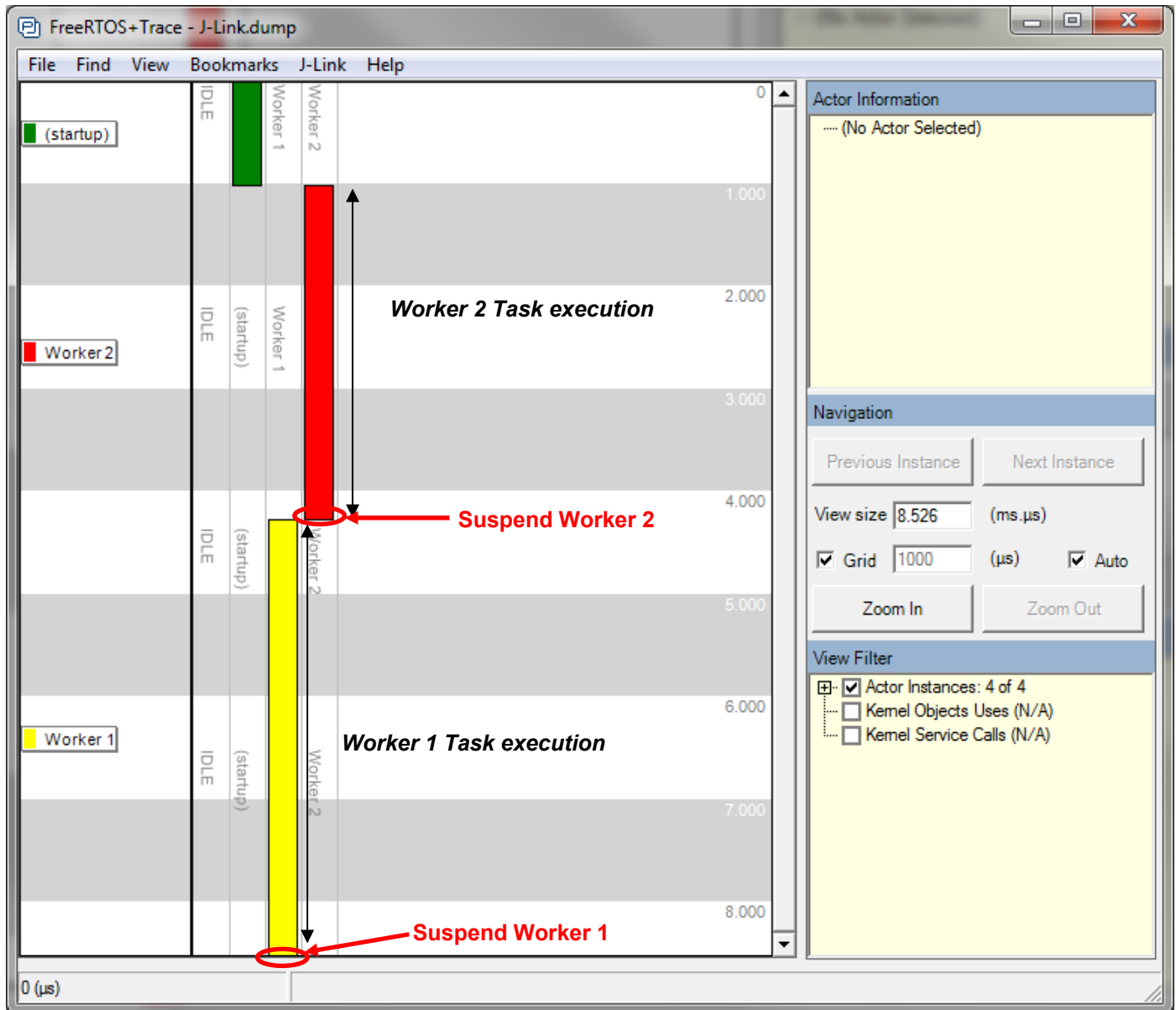
Schedule “Worker 2” task with higher priority than “Worker 1” task.

- In main.c file modify the tasks priority as following:  
    “Worker 1” task priority = 1  
    “Worker 2” task priority = 2
- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS + Trace output by clicking on “**J-Link>Read Trace**”



## RESULT




You should see the following result on “FreeRTOS + Trace”.





## TO DO

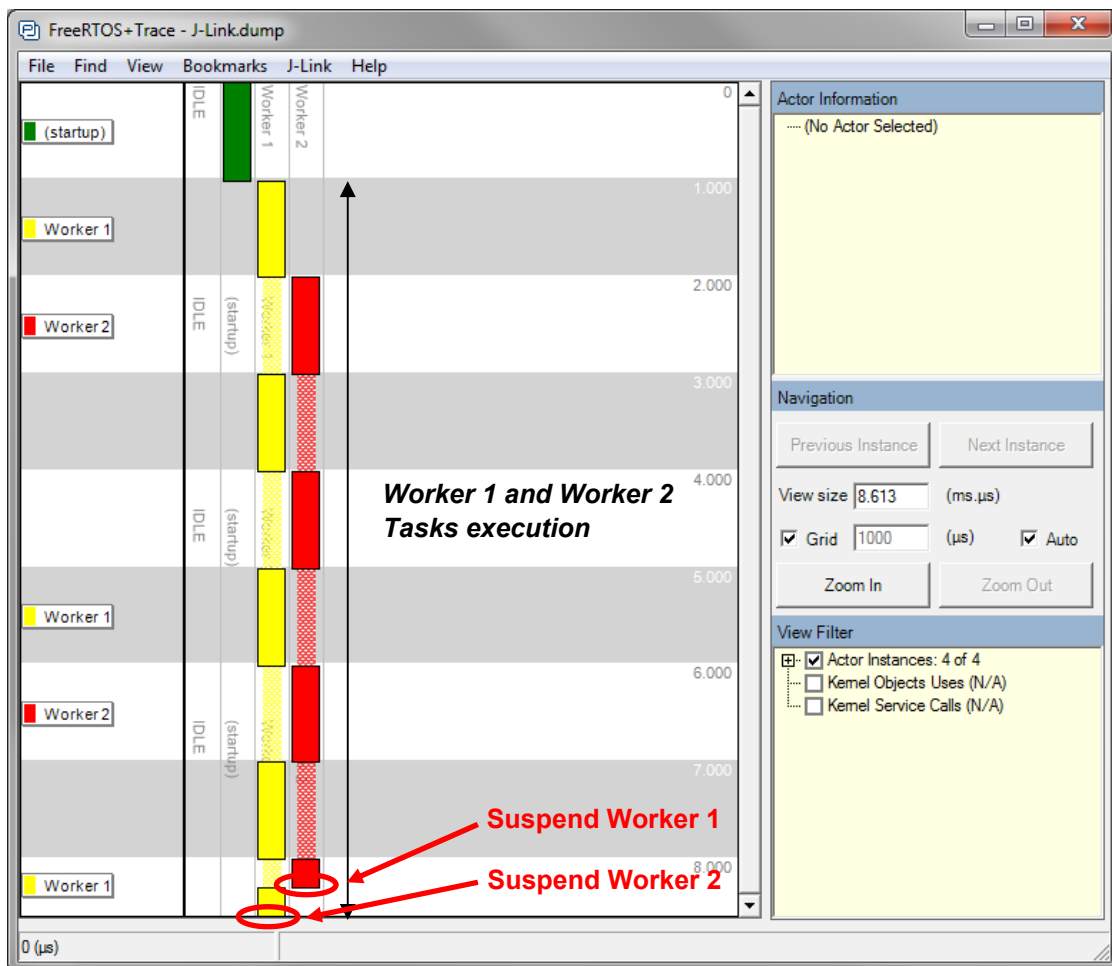
Schedule “Worker 2” task with the same priority as “Worker 1” task.

- In main.c file modify the tasks priority as following:
  - “Worker 1” task priority = 1
  - “Worker 2” task priority = 1
- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS+Trace output by clicking on “J-Link>Read Trace”



## RESULT

You should see the following result on “FreeRTOS+Trace”.



When two or more tasks share the same priority, the scheduler will cut their execution in time slice of one tick period. This is usually known as **Round Robin**.



**WARNING** Set back the task priority as following in order to ease comprehension for next assignment.

- “Worker 1” task priority = 2
- “Worker 2” task priority = 1

## 6. Assignment 3: Kernel Object Usage

In addition to standard task scheduling and management functionality, FreeRTOS provides Kernel Object, which allows tasks to interact with each other.

In this assignment we will cover the following topics:

- Software Timer
- Semaphores
- Queues

### 6.1 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 an xBlockTime, which is 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 functions:

- **xTimerCreate:**

**Description:** Function used to create a Timer Object

**Prototype:** `xTimerHandle xTimerCreate( const signed char *pcTimerName, portTickType xTimerPeriodInTicks, unsigned portBASE_TYPE uxAutoReload, void * pvTimerID, tmrTIMER_CALLBACK 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:**

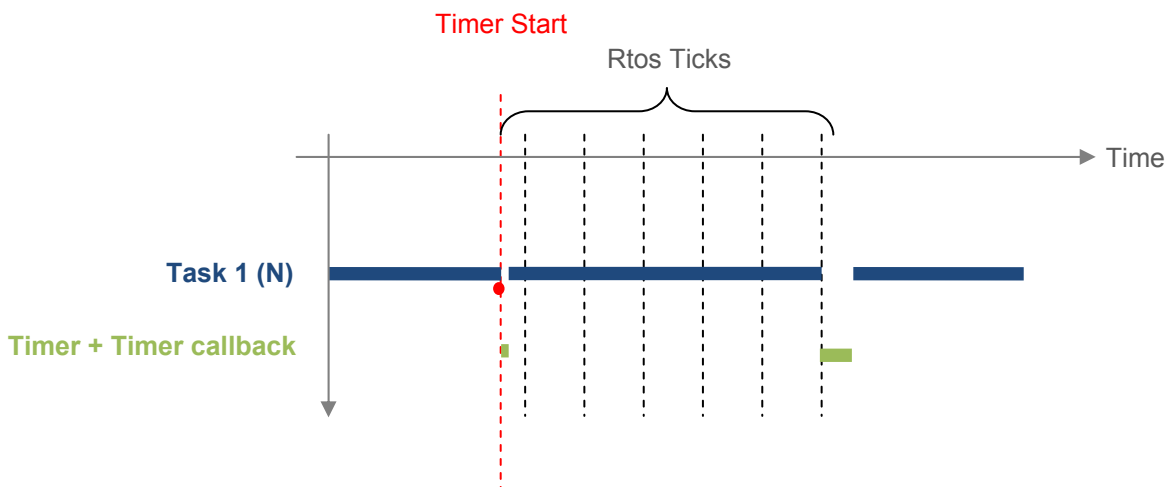
**Description:** Function used to stop a Timer

**Prototype:** `void xTimerStop( xTimer, xBlockTime )`

**Parameters:**

**xTimer:** targeted timer ID  
**xBlockTime:** Timeout for function to be handled by timer object

**Example five ticks timer with highest priority:**







## TO DO

Add a 50 ticks Software timer.

In this assignment, we will add a 50 ticks timer to our project.

- Configure the FreeRTOS kernel to use a software timer by modifying the following definition in the FreeRTOSConfig.h file:

```
/* Software timer definitions. */  
#define configUSE_TIMERS 1
```

- Define a timer handler at the beginning of your main.c file




```
xTimerHandle Timer_handle;
```

- Add the following “TimerCallback” function in your main.c file

```
void TimerCallback( xTimerHandle pxTimer )  
{  
    /* keep this empty for the moment.*/  
}
```

- Create and start a 50 ticks timer by using the “xTimerCreate” and “xTimerStart” functions in the main routine. These functions should be called before the existing *vTaskStartScheduler* function call.

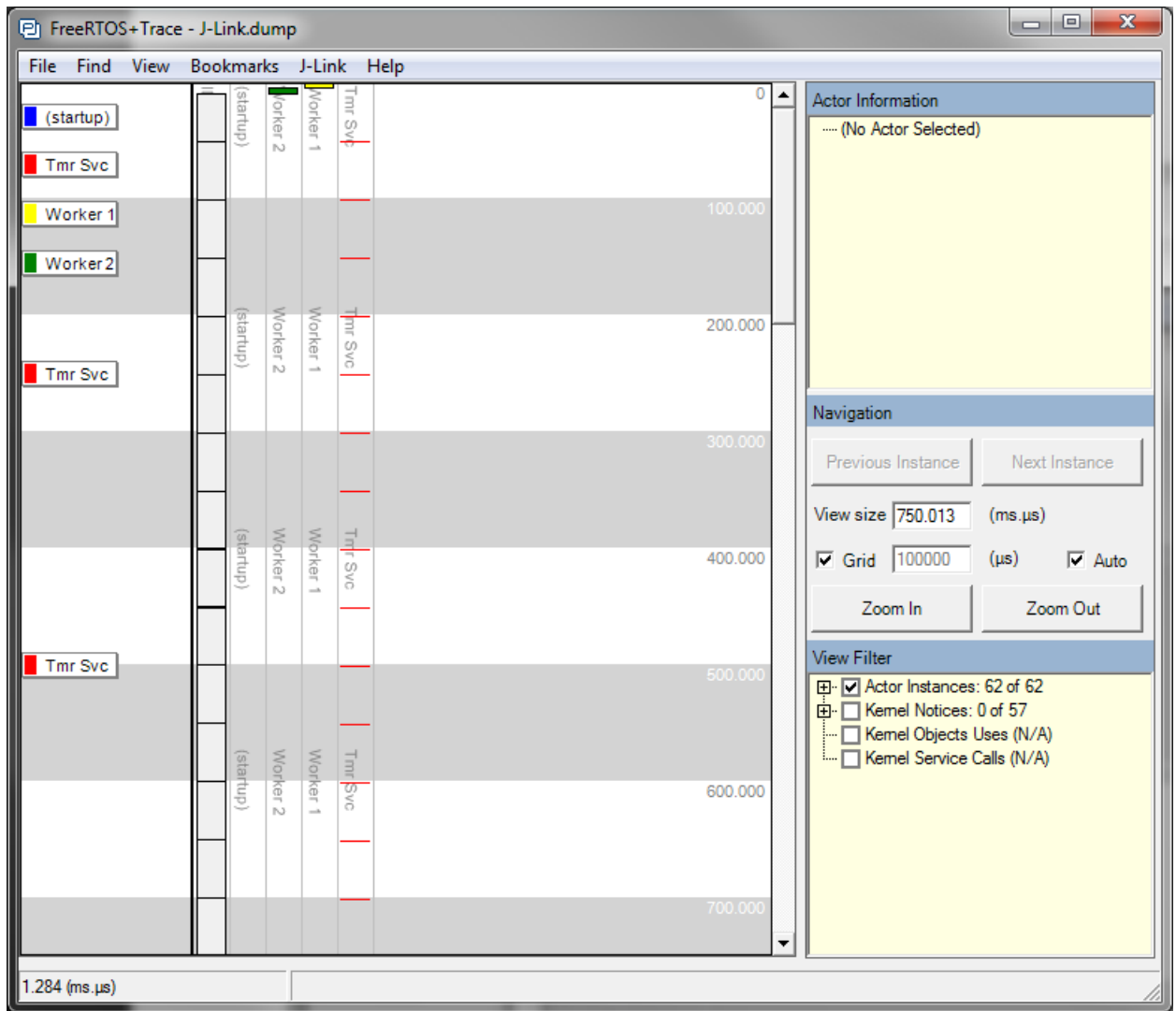
```
Timer_handle = xTimerCreate("Timer", 50, pdTRUE, 0, TimerCallback);  
xTimerStart( Timer_handle, 0);
```

- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS + Trace output by clicking on “**J-Link>Read Trace**”



## RESULT

You should see the following result on “FreeRTOS + Trace”:

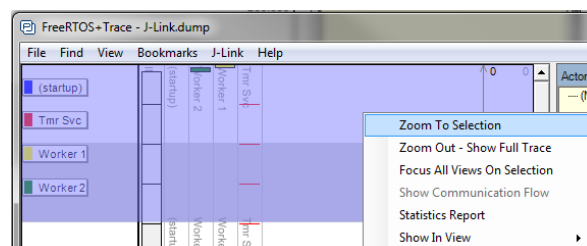


We can see on this trace the two worker tasks execution at the very beginning and the timer callback function executed every 50 ticks.



## TIPS

Under FreeRTOS+Trace, you can zoom on a specific part of the graphical trace representation, by just highlight the part of the graph to zoom in and right click on “Zoom to Selection”.



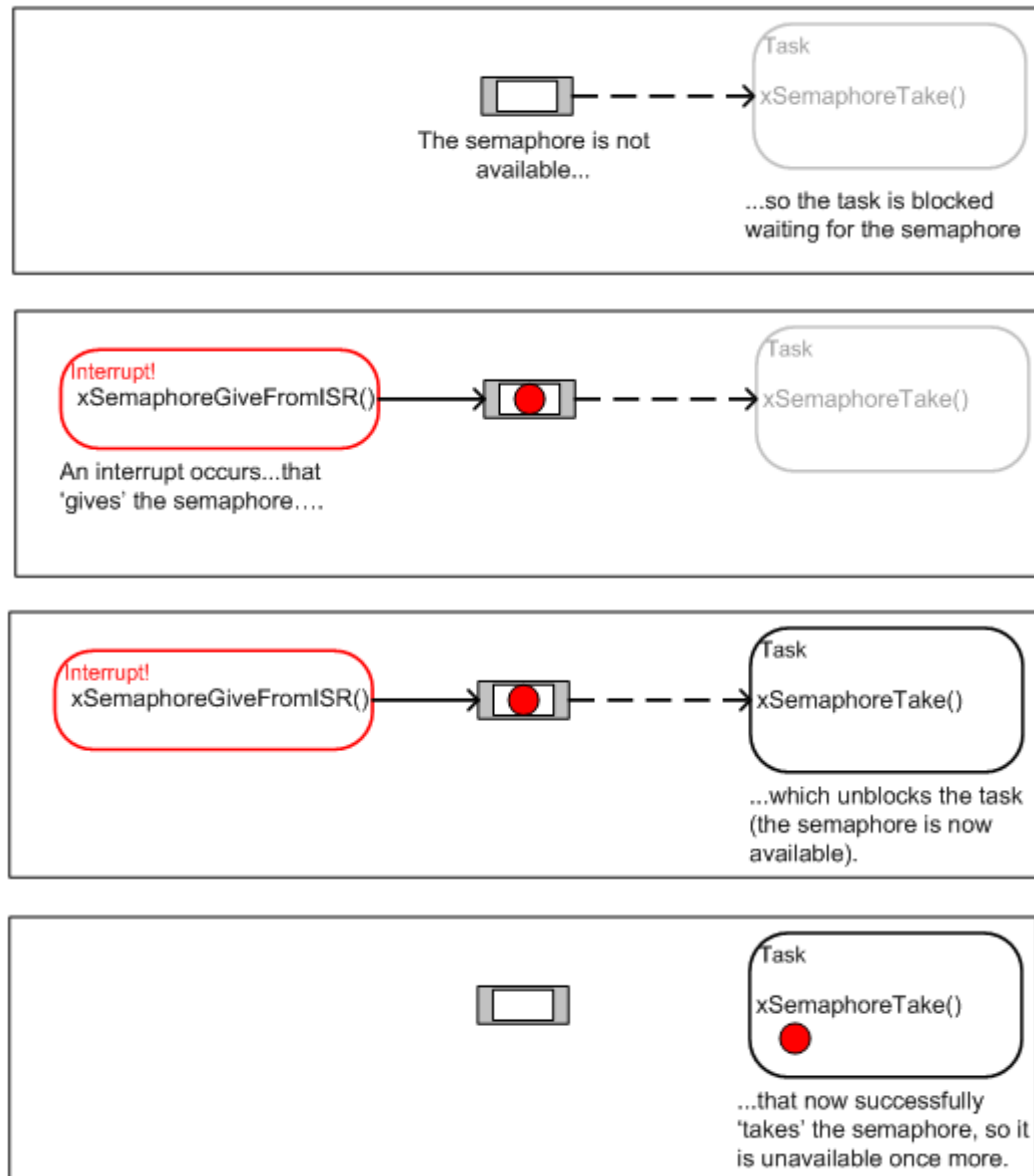
## 6.2 Semaphore Usage

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:



Here is a list of the Kernel functions that allows handling of semaphore:

- **vSemaphoreCreateBinary:**

**Description:** Function used to create a new binary semaphore

**Prototype:** `vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )`

**Parameters:**

- `xSemaphore`: Handle to the created semaphore. Should be of type `xSemaphoreHandle`

- **vQueueAddToRegistry:**

**Description:** Function used to Add queue/semaphore in the system registry and name it

**Prototype:** `vQueueAddToRegistry(xQueueHandle xQueue,  
char *pcQueueName );`

**Parameters:**

- `xQueue`: The handle of the queue/semaphore being added to the registry.
- `pcQueueName`: The name to be assigned to the queue. This is just a text string used to facilitate debugging.

- **xSemaphoreTake:**

**Description:** Function use to take a semaphore

**Prototype:** `xSemaphoreTake(xSemaphoreHandle xSemaphore,  
portTickType xBlockTime )`

**Parameters:**

- `xSemaphore`: A handle to the semaphore being taken - obtained when the semaphore was created.
- `xBlockTime`: The time in ticks to wait for the semaphore to become available. The macro `portTICK_RATE_MS` can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

- **xSemaphoreGive:**

**Description:** Macro to release a semaphore

**Prototype:** `xSemaphoreGive( xSemaphoreHandle xSemaphore )`

**Parameters:**

- `xSemaphore`: A handle to the semaphore being released obtained when the semaphore was created.

In our application we will use a semaphore in order to synchronize a “Manager task” with the previously created timer. The Manager task to be created will have the highest priority of the system but will need a semaphore from the Timer task in order to be unblocked. This manager task function will be to resume the worker tasks.



## TO DO

Create a manager task to synchronize with the software timer.

- Define as global a semaphore handler at the beginning of your main.c file

```
xSemaphoreHandle notification_semaphore;
```

- Implement the following function in your main.c file

```
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);




    /* Producer task Loop. */
    while (1)
    {
        /* Try to take the semaphore. */
        /* The lock is only released in the Timer callback function */
        if (xSemaphoreTake(notification_semaphore, 10000)) {
            vTaskResume(worker1_id);
            vTaskResume(worker2_id);
        }
    }
}
```

- Add the following task creation in the main routine. (Statement to be placed before vTaskStartScheduler(); function)

```
xTaskCreate(manager_task, "manager", configMINIMAL_STACK_SIZE+100, NULL,
tskIDLE_PRIORITY+3, NULL);
```

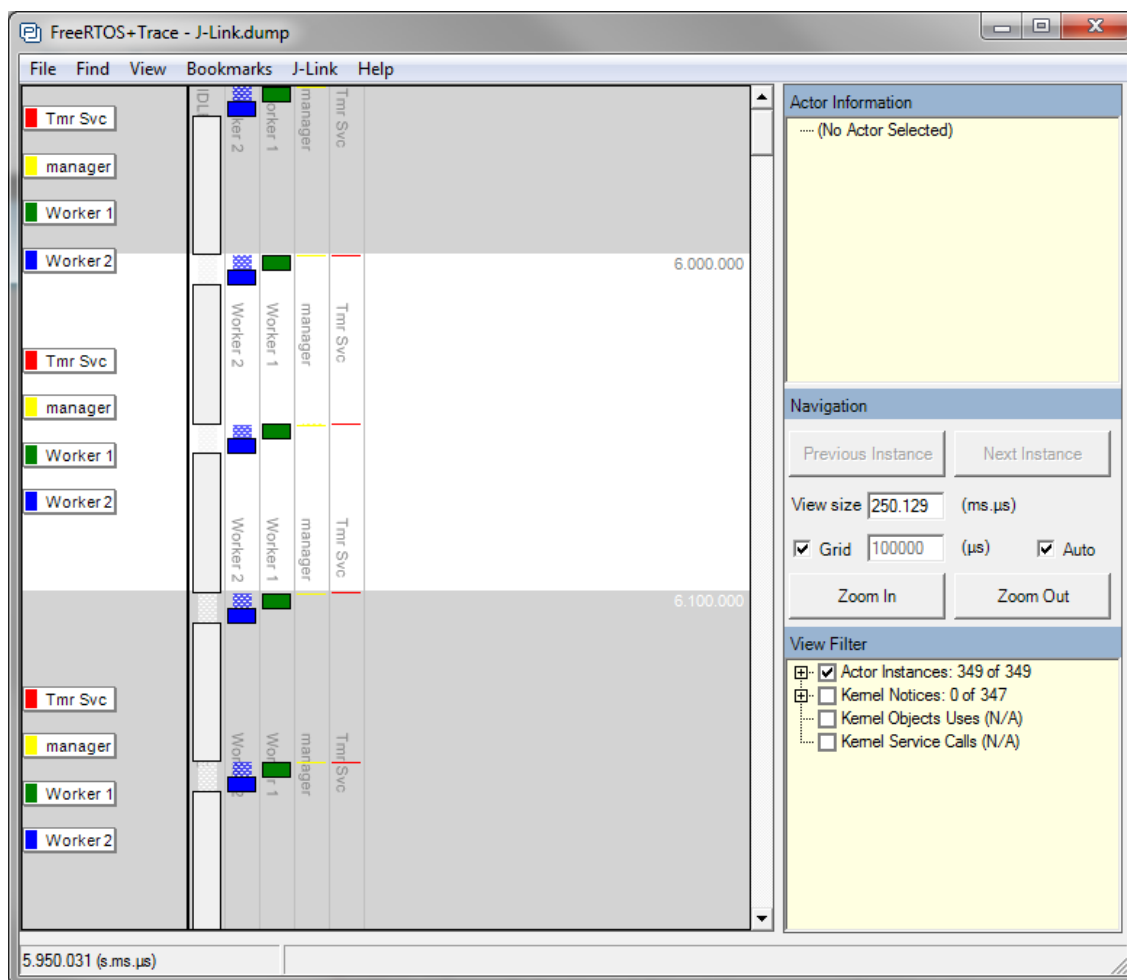
- Implement the following code in the Timer callback function

```
void TimerCallback( xTimerHandle pxTimer )
{
    /* Notify Manager task to start data processing. */
    xSemaphoreGive(notification_semaphore);
}
```

- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS + Trace output by clicking on “J-Link>Read Trace”



**RESULT** You should see the following result on “FreeRTOS + Trace”.



We can now see that manager task execution is synchronized with the Timer callback function and that Worker tasks are resumed just after manager task execution.



#### TIPS

In the 30 days evaluation version of FreeRTOS+Trace you can enable the display of Kernel objects uses by clicking on the Kernel Objects Uses checkbox. This option is located in the “View Filter frame” in the bottom right of FreeRTOS+Trace.

### 6.3 Queue Management

Queues are used for inter-task communication and synchronization in a FreeRTOS environment.

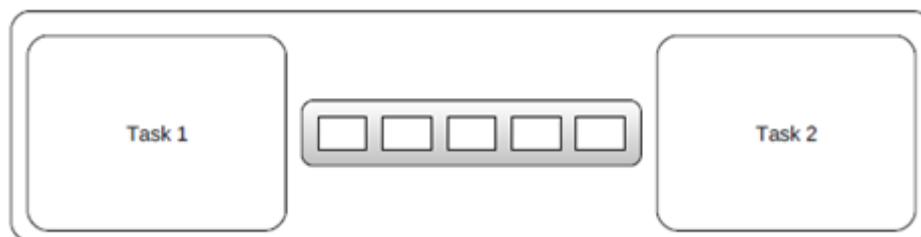
They are an important subject to understand as it is unavoidable to be able to build a complex application with tasks interacting with each other. They are meant to store a finite number of fixed size data.

Queues should be accessible for reads and writes by several different tasks, and don't belong to any tasks in particular.

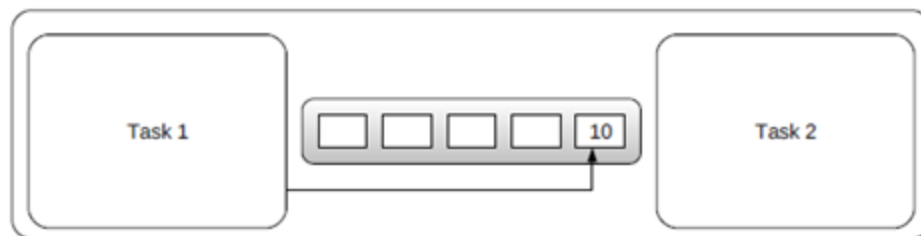
A queue is normally a FIFO, which means elements are read in the order they have been written. This behavior depends on the writing method: two writing functions can be used to write either at the beginning or at the end of this queue.

#### Illustration of queue usage:

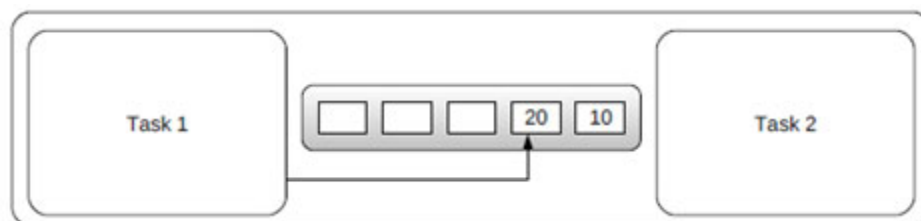
A queue is created to allow Task 1 and Task 2 to communicate. The queue can hold a maximum of five values. When a queue is created, it doesn't contain any values so is empty.



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.



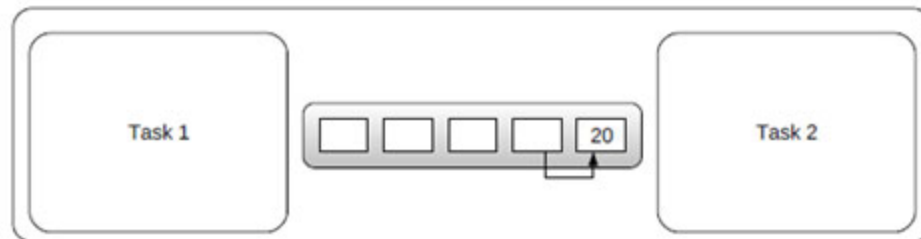
Task 1 sends another value. The queue now contains the previously written value and this newly added value. The previous value remains at the front of the queue while the new one is now at its back. Three spaces are still available.



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 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





## TO DO

Use message Queue to pass information from manager to worker tasks.

In this part, we will pass a Delay information from manager to Worker task using a message queue.

The manager will write two Delay values in the queue and each worker will pick-up one value and modify their Delay loop.

- Define a new Semaphore object as global in your main.c file

```
xQueueHandle Queue_id;
```

- Create the message queue by adding the following call to xQueueCreate function in your main routine. (The following line should be added before vTaskStartScheduler function call.)

```
Queue_id = xQueueCreate(2,sizeof (uint32_t));
```

- Modify the manager task to add Delay information to send to worker in the queue

```
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);

    /* Producer task Loop. */
    while (1)
    {
        /* Try to get the lock. */
        /* The lock is only released in the TC0 interrupt handler. */
        if (xSemaphoreTake(notification_semaphore, 10000)) {
            xQueueSend(Queue_id,&Delay1,0);
            xQueueSend(Queue_id,&Delay2,0);
            vTaskResume(worker1_id);
            vTaskResume(worker2_id);
        }
    }
}
```

- Comment the Delay setting in the Workers tasks




```
static void worker1_task(void *pvParameters)
{
    static uint32_t idelay, 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 , 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);
}
```

- Modify the two worker tasks to get Delay information from the queue

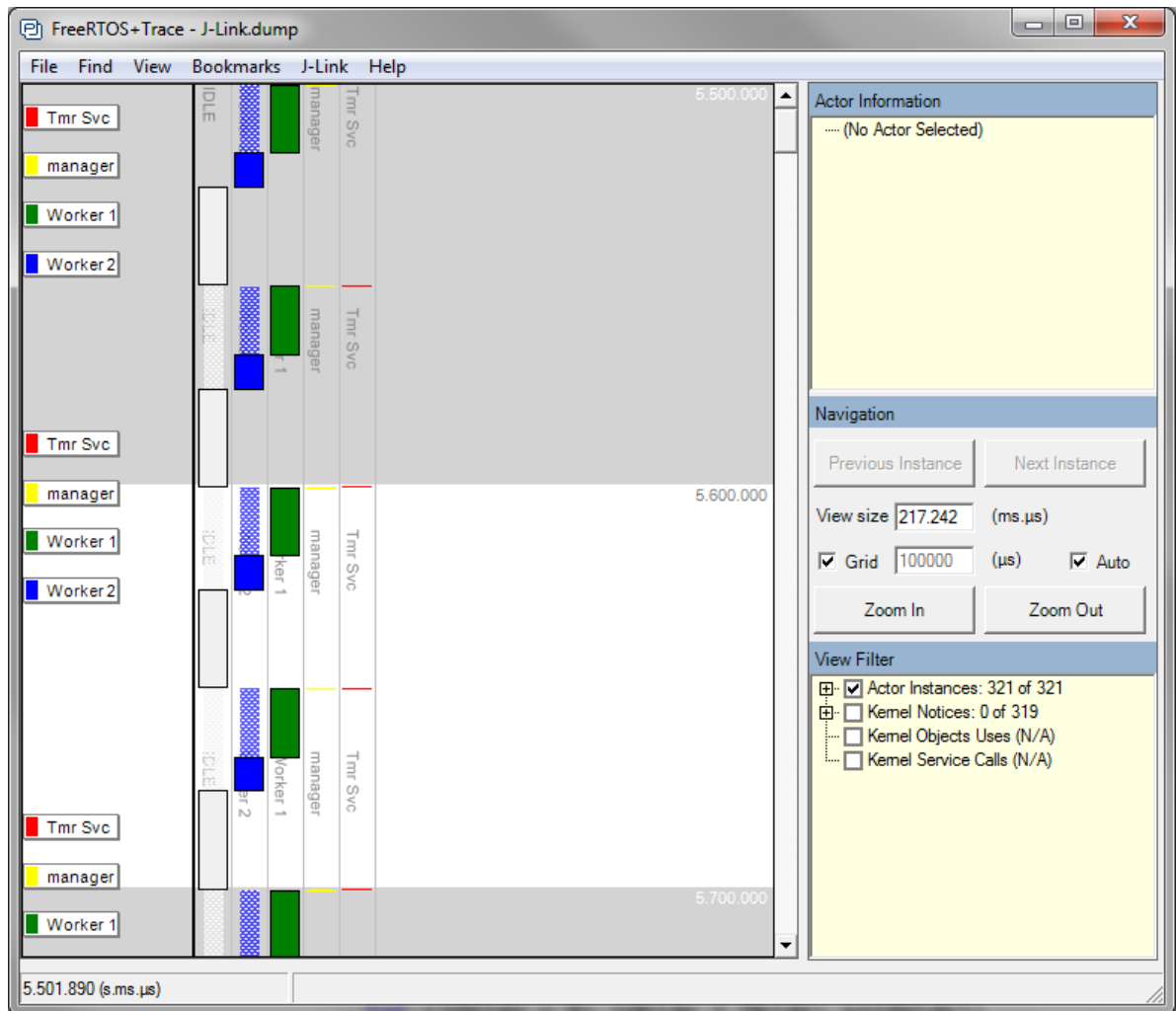
```
static void worker1_task(void *pvParameters)
{
    static uint32_t idelay, Delay ;
    //Delay = 100000;
    xQueueReceive(Queue_id, &Delay, 100000);
    /* Worker task Loop. */
    for(;;)
    {
        /* Simulate work */
        for (idelay = 0; idelay < Delay; ++idelay);
        /* Suspend Task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

static void worker2_task(void *pvParameters)
{
    static uint32_t idelay , Delay;
    //Delay = 100000;
    xQueueReceive(Queue_id, &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);
}
```

- Click on “Build” button:  and check the build log in the log output frame
- Click on “Start Debugging” button  to Download and run the program from internal Flash of the SAM4L
- Click on stop debugging button  in order to stop the debug session
- Refresh the FreeRTOS + Trace output by clicking on “**J-Link>Read Trace**”



**RESULT** You should see the following result on “*FreeRTOS+Trace*”.



On this graphical trace, we can see that Worker tasks length has changed according to information sent by manager task. This shows that our message queue is correctly working.

## 7. Conclusion

In this hands-on we have seen the basic functionality of the FreeRTOS Real Time Operating system and experimented on the following points:

- How to create and configure a FreeRTOS project under Atmel Studio 6
- How to make use of Graphical debugging tool
- How to make use of FreeRTOS basic functionality in an embedded project

This hands-on has given you the basic knowledge to develop your own Real-time application and understand the different FreeRTOS examples available in ASF.

## 8. Revision History

Doc. Rev.	Date	Comments
42247A	02/2014	Initial document release



Enabling Unlimited Possibilities®

**Atmel Corporation**

1600 Technology Drive  
San Jose, CA 95110  
USA

**Tel:** (+1)(408) 441-0311

**Fax:** (+1)(408) 487-2600

[www.atmel.com](http://www.atmel.com)

**Atmel Asia Limited**

Unit 01-5 & 16, 19F  
BEA Tower, Millennium City 5  
418 Kwun Tong Road  
Kwun Tong, Kowloon  
HONG KONG

**Tel:** (+852) 2245-6100

**Fax:** (+852) 2722-1369

**Atmel Munich GmbH**

Business Campus  
Parkring 4  
D-85748 Garching b. Munich  
GERMANY

**Tel:** (+49) 89-31970-0

**Fax:** (+49) 89-3194621

**Atmel Japan G.K.**

16F Shin-Osaki Kangyo Bldg.  
1-6-4 Osaki, Shinagawa-ku  
Tokyo 141-0032  
JAPAN

**Tel:** (+81)(3) 6417-0300

**Fax:** (+81)(3) 6417-0370

© 2014 Atmel Corporation. All rights reserved. / Rev.: 42247A - 02/2014

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation or its subsidiaries. ARM® and Cortex® are registered 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.