1 FreeRTOS on AVR32

FreeRTOS is a portable open source real-time kernel which operate in preemptive or cooperative mode. It has a small memory footprint (approx. 5-10 kb) and fairly short processing overhead. The features of the kernel is configured at compiletime, for instance the tick frequency and which kernel functions to include/exclude.

Tasks and semaphores are created at runtime, as well as the priority based scheduling.

1.1 Create a project including freeRTOS

FreeRTOS is partly supported by the Atmel Software Framework. However you can not include it using the ASF wizard. Atmel provides example projects for our board/processor combination which include freeRTOS. The procedure to create one such project which we will use a basis for this assignment is shown below.

1. Create a new "Example Project" by clicking on "New -> Example Project".

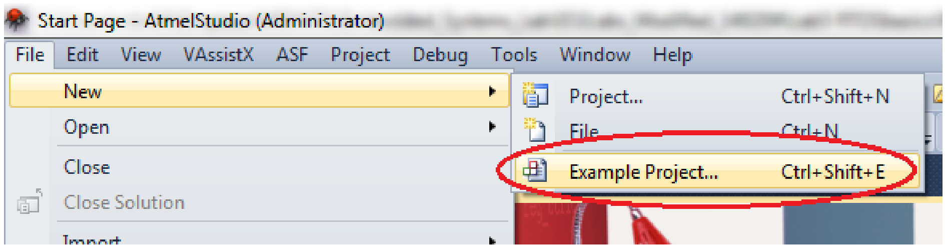
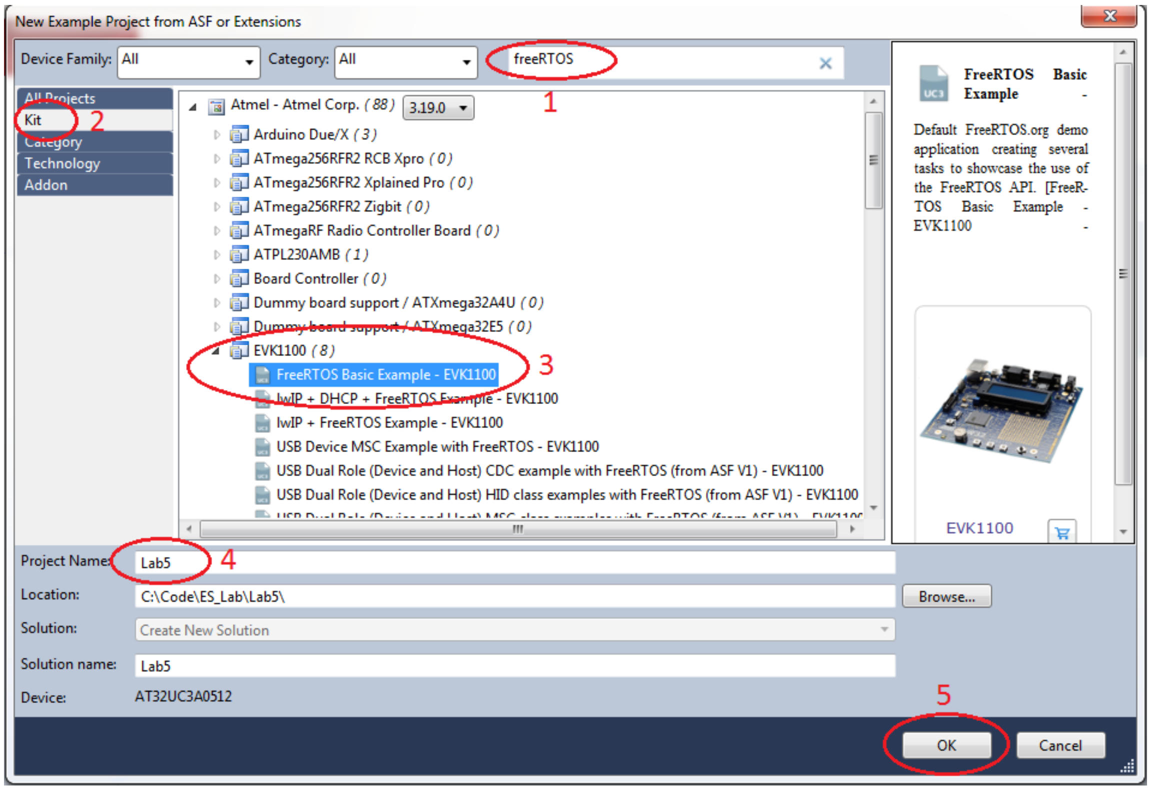


Figure 1: Create a new example project

- The window "New Example Project from ASF or Extensions" will appear.
  1. Write "freeRTOS" in the search field.
  2. Select "Kit" on the right, since we want examples targeting the EVK1100.
  3. Search for the "EVK1100" here you should find the example "FreeRTOS Basic Example - EVK1100" select it.
  4. Assign a project name and location as you know it already.
  5. Click "OK".
- You need to accept the freeRTOS license agreement.
- The project will show up in the "Solution Explorer". Since it is an example project the structure is a bit different than what you already know.
- The freeRTOS sources are included in "src -> ASF -> thirdparty -> freertos -> freertos-7.0.0".


New Example Project from ASF or Extensions

Device Family: All
Category: All
freeRTOS

All Projects
Kit
Category
Technology
Addon

Atmel - Atmel Corp. (88) 3.19.0
Arduino Due/X (3)
ATmega256RFR2 RCB Xpro (0)
ATmega256RFR2 Xplained Pro (0)
ATmega256RFR2 Zigbit (0)
ATmegaRF Radio Controller Board (0)
ATPL230AMB (1)
Board Controller (0)
Dummy board support / ATXmega32A4U (0)
Dummy board support / ATXmega32E5 (0)
EVK1100 (8)
FreeRTOS Basic Example - EVK1100
lwIP + DHCP + FreeRTOS Example - EVK1100
lwIP + FreeRTOS Example - EVK1100
USB Device MSC Example with FreeRTOS - EVK1100
USB Dual Role (Device and Host) CDC example with FreeRTOS (from ASF V1) - EVK1100
USB Dual Role (Device and Host) HID class examples with FreeRTOS (from ASF V1) - EVK1100

FreeRTOS Basic Example -

Default FreeRTOS.org demo application creating several tasks to showcase the use of the FreeRTOS API. [FreeRTOS Basic Example - EVK1100 -

EVK1100

Project Name: Lab5
Location: C:\Code\ES_Lab\Lab5\
Solution: Create New Solution
Solution name: Lab5
Device: AT32UC3A0512

Figure 2: Search for the freeRTOS project

- Since we created an example project but don't want to use the example we need to delete it that we are left with freeRTOS only.
- You can locate the "main.c" file of the project in "src -> ASF -> thirdparty -> demo -> avr32_uc3_example" as shown in the picture.
- We delete the whole "demo" folder.
- Create a new main file in the folder "src" and add the main function as you know it from previous labs.

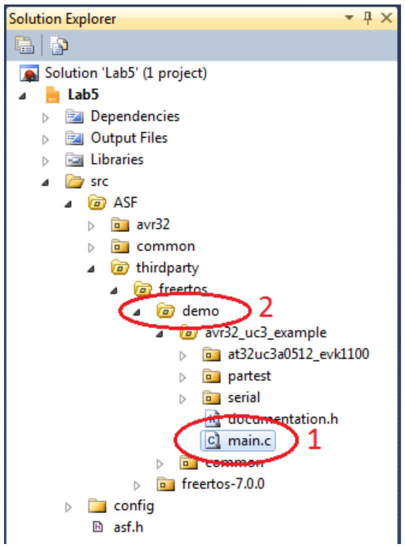


Figure 3: Search for the freeRTOS project

- At a minimum, include the following header files in your source code:
  - *FreeRTOS.h* - Needed for the freeRTOS kernel .
  - *task.h* - Needed for task creation.

1.2 FreeRTOS API

Detailed information of the freeRTOS API is found at *http://www.freertos.org* by clicking the left menu *API Reference.*

You are free to use all functions available in freeRTOS. To start with you should look at the functions listed below:

- *xTaskCreate()*
- *vTaskDelay()*
- *vTaskDelayUntil()*
- *xTaskGetTickCount()*
- *vTaskStartScheduler()*
- *taskYIELD()*
- *taskENTER_ CRITICAL()*
- *taskEXIT_ CRITICAL()*
- *taskENABLE_ INTERRUPTS()*
- *taskDISABLE_ INTERRUPTS()*

In later labs these _at least_ these functions will be needed:

- *vSemaphoreCreateBinary()*
- *xSemaphoreCreateMutex()*
- *xSemaphoreGive()*
- *xSemaphoreTake()*
- *xQueueCreate()*
- *vQueueDelete()*
- *xQueueReceive()*
- *xQueuePeek()*
- *xQueueSend()*

1.2.1 Good to know

Before reading the details about the functions, here are a collection of things good to know:

FreeRTOS uses a prefix to distinguish between different types. You don't have to follow the freeRTOS convention, but knowing what the strange letters are make things easier to understand. Any combination of these letters are valid, e.g. a function/variable named *pvParameters* is most problably of the type *void ** if it follow the freeRTOS convention.

- Variables
  - Variables of type *char* are prefixed *c*
  - Variables of type *short* are prefixed *s*
  - Variables of type *long* are prefixed *l*
  - Variables of type *float* are prefixed *f*
  - Variables of type *double* are prefixed *d*
  - Enumerated variables are prefixed *e*
  - Other types (e.g. structs) are prefixed *x*
  - Pointers have an additional prefixed *p*, for example a pointer to a short will be prefixed *ps*
  - Unsigned variables have an additional prefix *u*, for example an unsigned short be prefixed *us*
  - void functions are prefixed *v*
- Functions
  - Private functions are prefixed with *prv*
  - API functions are prefixed with their return type, as per the convention defined for variables
  - After the type prefix, function names start with the file in which they are defined. For example *vTaskDelay()* is defined in the file *tasks. c*
- malloc() and free()
  - When the real time kernel requires RAM, instead of calling *malloc()* it makes a call to *pvPortMalloc()*. When RAM is being freed, instead of calling *free()* the real time kernel makes a call to *vPortFree()*. The *malloc()* and *free()* calls are not deterministic.
- *portBASE_ TYPE* is the base type of the AVR32UC3 port, which is 32-bit signed long.
- *portTickType* is the type of real-time ticks and is 32-bit unsigned long.

1.3 FreeRTOSconfig.h

Configuration of freeRTOS is done through the *FreeRTOSconfig.h* file found in the *{project}/src/CONFIG* directory. Important configuration settings are breifly described below:

- *configUSE_PREEMPTION:* If set to 0 this configuration option choose the cooperative RTOS kernel. If set to 1 the preemptive kernel is choosen.
- *configCPU_CLOCK_HZ*: This need to be set to the internal MCU clock, 115200 Hz. Or you need to include include the Software Framework PM power manager and the switch to the external 12 Mhz crystal to use the default *FOSC0* frequency:

  ```
  pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
  ```
- *configTICK_RATE_HZ*: This is the frequency at which the RTOS tick will operate. As the tick frequency goes up, the kernel will become less efficient since it must service the tick interrupt service request (ISR) more often.
- *configMAX_PRIORITIES*: The total number of priority levels that can be assigned to a task. Each new priority level consume some memory. It is good to keep the priority levels at a minimum. If you have five tasks that run at different prioritys, set this value no less than 6 to include the idle task. If you try to create a task with higher priority than you have configured the freeRTOS kernel for, that task will be set to the maximum priority available to the kernel without warning.
- *configMAX_TASK_NAME_LEN*: The maximum number of characters that can be used to name a task. The name is mostly used for debugging and visualization of the system.
- *configIDLE_SHOULD_YIELD*: This determine if a task set at priority 0 (the idle task priority) should yield or not, to protect it from starvation.
- *configUSE_TRACE_FACILITY*: The FreeRTOS core has trace functionality built in. This item is set to 1 if a kernel activity trace is desired. The trace log is created in RAM so a buffer needs to be allocated.
- *configMINIMAL_STACK_SIZE*: This is the stack size used by the idle task. An analysis of the optimal value should be possible. Sticking to the default value should be good enough for our purposes.
- *configUSE_CO_ROUTINES*: Defaults to 0. We do not use co-routines in these labs.

1.4 Tasks and the real-time scheduler

The scheduler algorithm of freeRTOS is simply "highest priority goes first". A low priority task do not execute until there are no higher priority tasks ready for execution. If a higher priority task is always ready, the lower priority task never execute. This is called starvation.

When more than one task have the same priority, tasks are executed in a round robin fashion. Preemption is only allowed if the *configUSE_PREEMPTION* is set to 1 otherwise the cooperative scheduler will be used.

Before starting the real-time scheduler at least one task must be created. This is done by the *xTaskCreate()* function. The idle task, which mostly does not do anything, is created automatically at the same time. It have a priority of *tskIDLE_PRIORITY*, which is set to 0, no other task can have lower proirity than that.

1.4.1 Task creation

A task should have the following structure, particulary it should never be allowed to return:

```
void vTaskFunction( void * pvParameters )
{
    ...
    for(;;) {
        /* Task application code below. */
        ...
    }
}
```

The task is created by a call to the xTaskCreate() function:

```
xTaskCreate(pdTASK_CODE pvTaskCode,
            const signed portCHAR * const pcName,
            unsigned portSHORT usStackDepth,
            void * pvParameters,
            unsigned portBASE_TYPE uxPriority,
            xTaskHandle * pxCreatedTask );
```

Parameters:

- *pvTaskCode* - Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- *pcName* - A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN.
- *usStackDepth* - The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the

stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.

- *pvParameters* - Pointer that will be used as the parameter for the task being created.
- *uxPriority* - The priority at which the task should run.
- *pvCreatedTask* - Used to pass back a handle by which the created task can be referenced.

Returns:
pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs. h

1.4.2 Starting the scheduler

The requirement for starting the scheduler is at least one task have previously been created.

```
void vTaskStartScheduler(void);
```

Example:

```
// Task to be created.
void vTaskCode(void * pvParameters)
{
    for(;;) {
        // Task code
        ...
    }
}

// Function that creates a task.
void vOtherFunction(void)
{
    static unsigned char ucParameterToPass;
    xTaskHandle xHandle;

    // Create the task, store the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 MINIMAL_STACK_SIZE,
                 &ucParameterToPass,
                 tskIDLE_PRIORITY + 1,
                 &xHandle );

    vTaskStartScheduler();
    ...
}
```

2 Serial I/O API in freeRTOS

Two channels are available for serial output, one is used for debug messages and the other for normal serial communication. To conform with freeRTOS we configured the serial output channels in the *freeRTOSconfig.h* file and set both to USART1:

```
...
#define configDBG                       1
#define configDBG_USART                 (&AVR32_USART1)
#define configDBG_USART_RX_PIN          AVR32_USART1_RXD_0_0_PIN
#define configDBG_USART_RX_FUNCTION     AVR32_USART1_RXD_0_0_FUNCTION
#define configDBG_USART_TX_PIN          AVR32_USART1_TXD_0_0_PIN
#define configDBG_USART_TX_FUNCTION     AVR32_USART1_TXD_0_0_FUNCTION
#define configDBG_USART_BAUDRATE        57600
#define serialPORT_USART                (&AVR32_USART1)
#define serialPORT_USART_RX_PIN         AVR32_USART1_RXD_0_0_PIN
#define serialPORT_USART_RX_FUNCTION    AVR32_USART1_RXD_0_0_FUNCTION
#define serialPORT_USART_TX_PIN         AVR32_USART1_TXD_0_0_PIN
#define serialPORT_USART_TX_FUNCTION    AVR32_USART1_TXD_0_0_FUNCTION
#define serialPORT_USART_IRQ            AVR32_USART1_IRQ
#define serialPORT_USART_BAUDRATE       57600
...
```

2.1 Serial initialization

The serial port must be initialized before use. How this is done is shown in the example below.

2.2 Serial output

Output is performed by the function *usart_write_line()* which tries to send a string to the USART for *USART_DEFAULT_TIMEOUT* times before giving up. *USART_DEFAULT_TIMEOUT* is set by default to 10000.

```
void usart_write_line(volatile avr32_usart_t * usart,
                      const char * string);
```

Parameters:

- *usart* - The USART port, e.g. serialPORT_USART
- *string* - Pointer to the string to output

Returns:

- *USART_SUCCESS* on success
- *USART_FAILURE* on failure

2.3 Serial input

Input is done by the function *usart_read_char()* which check the RX buffer for a recieved char and copies it to a variable.

```
int usart_read_char(volatile avr32_usart_t * usart, int * ch);
```

Parameters:

- *usart* - The USART port, e.g. serialPORT_USART
- *ch* - Pointer to the where the read character should be stored

Returns:

- *USART_SUCCESS* - The character was read successfully
- *USART_RX_EMPTY* - The RX buffer was empty
- *USART_RX_ERROR* - An error was deteceted

Shown below is how the USARTs are initialized and used:

```
#include "board.h"
#include "compiler"
#include "gpio.h"
#include "pm.h"
#include "usart.h"
...
void init_usart(void)
{
    static const gpio_map_t USART_SERIAL_GPIO_MAP =
    {
        { serialPORT_USART_RX_PIN, serialPORT_USART_RX_FUNCTION },
        { serialPORT_USART_TX_PIN, serialPORT_USART_TX_FUNCTION }
    };

    static const gpio_map_t USART_DEBUG_GPIO_MAP =
    {
        { configDBG_USART_RX_PIN, configDBG_USART_RX_FUNCTION },
        { configDBG_USART_TX_PIN, configDBG_USART_TX_FUNCTION }
    };

    static const usart_options_t USART_OPTIONS =
    {
        .baudrate     = 115200,
        .charlength   = 8,
        .paritytype   = USART_NO_PARITY,
        .stopbits     = USART_1_STOPBIT,
        .channelmode  = USART_NORMAL_CHMODE
    };

    pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
    gpio_enable_module(USART_SERIAL_GPIO_MAP, 2);
    gpio_enable_module(USART_DEBUG_GPIO_MAP, 2);
    usart_init_rs232(serialPORT_USART, &USART_OPTIONS, FOSC0);
    usart_init_rs232(configDBG_USART, &USART_OPTIONS, FOSC0);
}

    ...
    /* Output a debug message on the debug USART */
    usart_write_line(configDBG_USART, "Serial initialized");
    usart_write_line(serialPORT_USART, "Press a key...");
    while (usart_read_char(serialPORT_USART, &c) == USART_RX_EMPTY);
    ...
```

3 Assignments

1. Write a program using freeRTOS that blink the LEDs at different intervalls, LED1 = 1 s, LED2 = 2 s etc. Pressing a button (1-3) should make the corresponding LED (1-3) stay lit for 10 seconds while the others keep blinking. Make the various task output their actions to the serial port.

2. How can you detect deadline misses? Write software functions to detect deadline misses. How can you test your software? Force a priority inversion to see the deadline miss.
   Note: You must discuss the design of this task with the Lab assistant and get approval before starting to code. The design should also be attached with the report.

   Important Tip: Read the study material provided in Lecture 8 before starting the design.

4 Report

The report should include the following.

1. What is the main difference between *preemptive* and *cooperative scheduling*?

2. What is the difference between *vTaskDelay()* and *vTaskDelayUntil()*?

3. Design of Assignment 2.

5 Optional Assignment

1. Measure the context switching time using a timer. How does it vary with the number of active tasks. Plot context switch time as a function of number of active tasks in a graph.