FreeRTOS Guide

Cross-Domain Development Kit XDK110
Platform for Application Development

Bosch Connected Devices and Solutions

BOSCH
Invented for life

XDK110: Data Sheet

Document revision	2.1
Document release date	05.10.17
Workbench version	3.0.0
Document number	BCDS-XDK110-GUIDE-FREERTOS
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. **Subject to change without notice**

XDK FreeRTOS Guide

PLATFORM FOR APPLICATION DEVELOPMENT

The XDK platform offers many useful high level APIs. When dealing with basic topics like task scheduling or intertask communication, though, we have to resort to the functions offered by the underlaying operating system. The operation system that powers the XDK is called FreeRTOS. It is a lightweight, open source **r**eal **t**ime **o**perating **s**ystem (RTOS), built specifically for embedded systems. This guide will give an overview of the FreeRTOS features available within the XDK SDK and will demonstrate how to use them in code. The guide will also cover the CmdProcessor, a utility module that provides simplified task scheduling by combining some of these features. For further documentation and code examples on FreeRTOS functionality, please refer to www.freertos.org.

Table of Contents

1. API OVERVIEW 3

2. API USAGE 4

- 2.1 PREPARATION 4
- 2.2 PREDEFINED CONSTANTS 5
- 2.3 TASKS 6
- 2.4 TIMERS 10
- 2.5 QUEUES 13
- 2.6 SEMAPHORES 16

3 CMDPROCESSOR 19

4 MEMORY MANAGEMENT 23

- 4.1 FLASH MEMORY 23
- 4.2 RANDOM ACCESS MEMORY (RAM) 25

5 DOCUMENT HISTORY AND MODIFICATION 27

This guide postulates a basic understanding of the XDK and the according Workspace. For new users we recommend going through the following tutorials first:

- *XDK Workbench First Steps* (https://xdk.bosch-connectivity.com/documents/37728/55858/XDK+First+steps/15c62c10-ad53-456f-b567-2f14facd4eb2)
- *Hello World implementation* (XDK User Guide Page page 46ff
  https://xdk.bosch-connectivity.com/community/-/message_boards/message/94206)

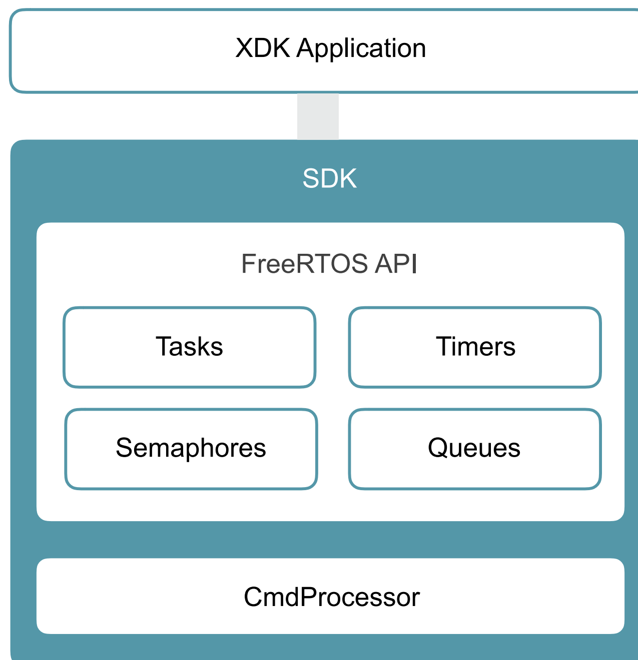
© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

1. API Overview

The FreeRTOS API is part of the XDK SDK and offers applications access to system level functionality. The API is separated into modules. Each module provides a number of functions and definitions related to a specific system-level construct. The following picture shows an overview of the available modules: tasks, timers, semaphores and queues.

The picture also shows the CmdProcessor module, which will be explained at the end of this guide. As depicted, this CmdProcessor is not part of the FreeRTOS API itself. Rather, it is a separate utility module that provides additional functionality on top of the FreeRTOS features. Like most higher level APIs, it is intended to offer a simple interface to otherwise complex logic.

Picture 1. Software Architecture



Each of these modules can be linked into an application by including the corresponding system header file.

A complete reference over all available functions, data types and constants can be found at http://xdk.bosch-connectivity.com/xdk_docs/html/group__freertos__api.html.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2. API Usage

This chapter will demonstrate how to use the data types and methods provided by the FreeRTOS API.

2.1 Preparation

The XDK-Workbench offers a sample project that is called `XdkApplicationTemplate`. It can be imported into the project list by clicking on Help > Welcome > XDKApplicationTemplate in the XDK-Workbench. This project contains the minimum setup for a any XDK application. The following snippet shows the function where control is handed over to the application code, also called the entry point of the application.

Note: `appInitSystem()` has to be the last function of the source file.

Code 1. XDK Application entry point

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    if (CmdProcessorHandle == NULL)
    {
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    BCDS_UNUSED(param2);

    // execution starts here
    doSomething();
}
```

`appInitSystem()` is the starting point of every XDK application. It receives two input parameters, a void pointer and an integer. The meaning of these parameters will be explained in section 2.7. In most cases, the code provided in the application template is sufficient and doesn't need further attention. Custom application code should start below these generated lines.

A simple application, that only needs to perform a single routine once, all the application code could be executed from within this function. However, most XDK applications need to repeat certain procedures at specific events (e.g. interrupt, resource becomes available) or times (e.g. every 100 ms). To achieve these kind of behaviours, FreeRTOS (like most operating systems) offers two low-level structures that allow the scheduled execution of code: tasks (see section 2.3) and timers (see section 2.4).

Note: Since it takes a few seconds to create a data connection between the PC and the XDK after the XDK is (re)started, text that is printed to the console during these first seconds of execution won't be visible in the Workbench console.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2.2 Predefined Constants

When using the FreeRTOS API, an application often needs to reference predefined or system-provided values like maximum task priority, minimum stack size, heap size, etc. These values are defined in a top-level header called `FreeRTOS.h`. By including that header, the application gains access to all required constants and definitions:

Code 2. Including the FreeRTOS header

```
#include "FreeRTOS.h"
```


© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2.3 Tasks

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task is executed within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executed at any point in time and the scheduler is responsible for deciding which task this should be. The scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the schedulers activity, it is the responsibility of the scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this, each task is provided with its own stack. When the task is swapped out, the execution context is saved to it's stack so it can be exactly restored when the same task is later swapped back in.

The size of a task's stack is allocated at the time of its creation and can't be modified later. Therefore, this size has to be big enough to contain the tasks complete state information at any point in time. If a tasks stack exceeds it's stack size, the XDK will throw a runtime error (a so called stack overflow). This should be avoided by choosing a suitable stack size.

An RTOS can multitask using these same principals - but their objectives are very different to those of non real time systems. The different objective is reflected in the scheduling policy. Real time / embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the system must respond. The RTOS scheduling policy must ensure these deadlines are met. To achieve this objective, FreeRTOS executes tasks according to their priority, which has to be set by the developer. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously.

The following code snippet shows how to create a task with minimum priority:

Code 3. Create a task

```
static xTaskHandle taskHandle = NULL; // globally accessible

void doSomething(void)
{
    BaseType_t result = xTaskCreate(
        myTaskFunction, // function that implements the task
        (const char * const) "My Task", // a name for the task
        configMINIMAL_STACK_SIZE, // depth of the task stack
        NULL, // parameters passed to the function
        tskIDLE_PRIORITY, // task priority
        &taskHandle // pointer to a handle for later reference
       );

    if(pdPASS != result) {
        assert(false);
    }
}
```

Note: You may have noticed that the task module wasn't included explicitly. This is because it is already included by the `FreeRTOS.h` header, as it is the most commonly used module.

`xTaskCreate` is used to create a task. Let's have a look at it's parameters:

Table 1. Parameters of xTaskCreate

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Argument	Description
Task Function	The function that implements the task. This function has to (1) accept a void pointer as its only parameter and (2) return void. See **Code 4** for an example.
Name	An arbitrary string that will be used as the name of the new task. This name will only be used internally or for debugging purposes. The maximum length of the string is 10.
Stack Size	An unsigned integer (16 bit) that defines the size of the stack that will be allocated for the new task. In **Code 3**, the system defined minimal stack size is used. For more complex tasks, this has to be adjusted.
Parameters	A pointer that will be provided as an argument to the task function, used to pass initial data to the new task. Can be `NULL`.
Priority	The priority of the new task. This value must be between `tskIDLE_PRIORITY` (0, lowest) and `configMAX_PRIORITIES` - 1 (4, highest). The priority `configMAX_PRIORITIES` itself is reserved for interrupt handling. As reference, the default priority for timers (see section 2.4) is 2.
Created Task	A pointer to a `xTaskHandle` that will store the created task object.

Note: The system constants for stack size and priority are defined in the file `FreeRTOSConfig.h`. Some default values of these parameters within the XDK stack can be found in the file `BCDS_TaskConfig.h`. Both files are part of the Configuration module.

The return code of `xTaskCreate` indicates whether the task was created successfully. If there wasn't enough memory left to allocate a stack with the requested size, the function returns `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`. If `pdPASS` is returned, the task was successfully created.

Note: The task function has to be declared before it can be used in `xTaskCreate`. This can be achieved by placing it above the code that creates the task (**Code 3**).

Code 4. Implementing a task

```
#define SECONDS(x) ((portTickType) (x * 1000) / portTICK_RATE_MS)

void myTaskFunction(void *pParameters)
{
    (void) pParameters;

    for (;;) {
       vTaskDelay(SECONDS(3)); // block this task for 3 seconds

       // then perform the task routine here
    }
}
```

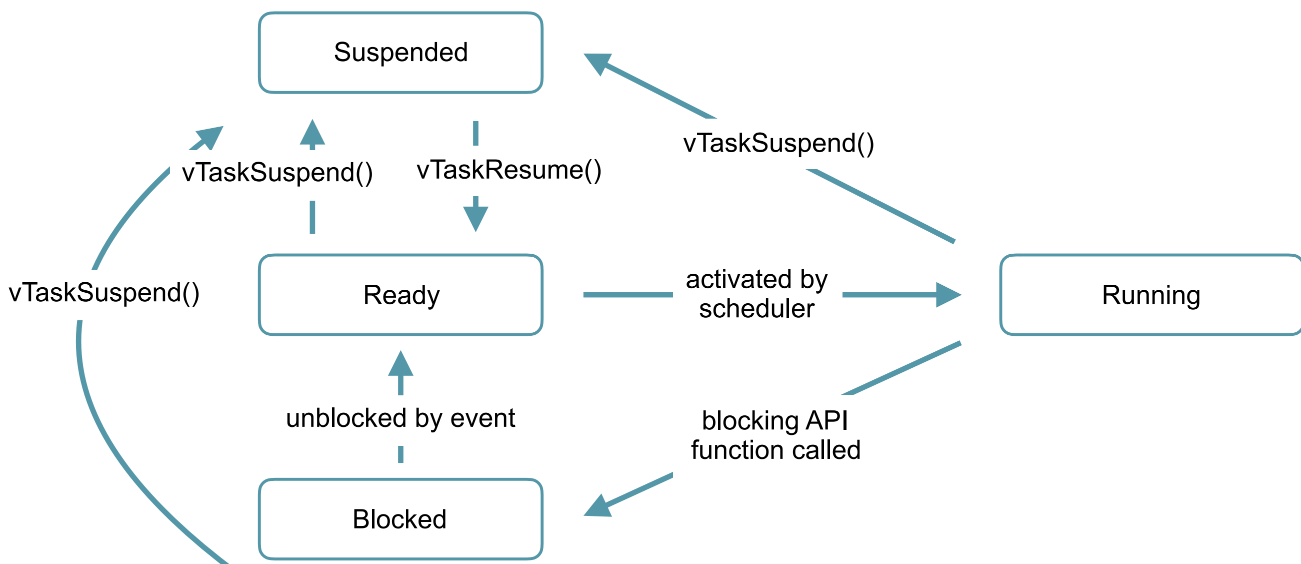

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

The function parameter will be filled with the data that was passed to `xTaskCreate` (in **Code 3**, this was `NULL`).

Usually, a task function will be implemented as an infinite loop that blocks until something happens, then does whatever it was intended to do and then blocks again. While the task is blocked, the operating system can use the available processing time to execute other tasks, or go into idle state if none are available. In the example in **Code 3**, `vTaskDelay` is used as a cheap way to put a task into blocked state. This basically creates a timer. There are, however, better ways to create a timer, which will be shown in section 2.4. More realistically, this statement would be a call that waits for incoming network requests, for a hardware interrupt or for a queue to provide some value.

Apart from being blocked, a task can have three other states, that are shown in **Picture 2**. Immediately after its creation, a task is put into ready state. As soon as the scheduler puts it into running state for the first time, the task function will be executed.

Picture 2. Task states



Apart from controlling the flow of a task from within this task, there are also functions that can modify a tasks state from other tasks:

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 5. Controlling tasks

```
xTaskHandle taskHandle = NULL;
// create task and put it into state READY
xTaskCreate(myTaskFunction, (const char * const) "My Task",
      configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, &taskHandle);
// ...
vTaskSuspend(taskHandle); // put task into state SUSPENDED
// ...
vTaskResume(taskHandle); // put task back into state READY
// ...
vTaskDelete(taskHandle);   // delete task permanently
// ...
vTaskDelete(NULL); // delete the CURRENT task
```

If you pass `NULL` instead of a task handle to one of the task controlling functions, it will affect the task that is calling the function!

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2.4 Timers

Timers allow to execute functions 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. Put simply, the timer's callback function is executed when the timer's period expires.

Timer callback functions execute in the context of the timer service task. It is therefore essential that timer callback functions never attempt to block. For example, a timer callback function must not call `vTaskDelay()` or specify a non zero block time when accessing a queue or a semaphore (see section 2.5 and 2.6).

The following code snippet shows how to create and start a timer:

Code 6. Create and start a timer

```
#include "timers.h" // include the timer module

#define MILLISECONDS(x) ((portTickType) x / portTICK_RATE_MS)
#define SECONDS(x) ((portTickType) (x * 1000) / portTICK_RATE_MS)

void doSomething(void)
{
    xTimerHandle timerHandle = xTimerCreate(
      (const char * const) "My Timer", // used only for debugging purposes
      SECONDS(12), // timer period
      pdFALSE, //Autoreload pdTRUE or pdFALSE - should the timer start again
               //after it expired?
      NULL, // optional identifier
      myTimerCallback // pointer to a static callback function
     );

     if(NULL == timerHandle) {
         assert(pdFAIL);
         return;
     }

     BaseType_t timerResult = xTimerStart(timerHandle, MILLISECONDS(10));

     if(pdTRUE != timerResult) {
         assert(pdFAIL);
     }
```


© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

`xTimerCreate` is used to create a timer. Let's have a look at it's parameters:

Table 2. Arguments of xTimerCreate

Argument	Description
Name	A string constant that will be used as the name of the new timer. This name will only be used internally or for debugging purposes. Any string could be chosen.
Period	The time interval after which the timer expires and the timer function is called.
Auto Reload	A boolean (`pdTRUE` or `pdFALSE`) that defines whether the timer should be automatically restarted after it expires.
ID	This argument can be used to assign an arbitrary value to the new timer in order to identify it at a later point, for example in a callback function that handles multiple timers.
Callback Function	A function (pointer) that is called when the timer expires. The function has to (1) be static, (2) accept a `xTimerHandle` as its only parameter and (3) return void. See **Code 7** for an example.

`xTimerCreate()` returns an `xTimerHandle` object or `NULL` if the timer couldn't be created. This can be the case either if the period was set to 0, or if there isn't enough heap space to allocate the timer structures.

If the timer was created successfully, it can be started with `xTimerStart()`. The first argument to this function is the timer handle, the second is the time interval that the calling task should be held in blocked state while the system tries to create the timer. The return code indicates whether the timer was started successfully.

The next snippet shows an example for a timer callback function:

Note: The timer callback function has to be declared before it can be used in `xTimerCreate()`. This can be achieved by placing it above the code that creates the timer (**Code 6**).

Code 7. Timer callback function

```
#include <stdio.h> // for printf

static void myTimerCallback(xTimerHandle xTimer)
{
    (void) xTimer;
    printf("Timer fired!\r\n");
}
```

FreeRTOS offers various functions to control and manipulate timers. They all expect as their last argument (like `xTimerStart()`) a time interval that defines how long the calling task is held in blocked state while the system tries to perform the respective command.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 8. Manipulating timers

```
#define MILLISECONDS(x) ((portTickType) x / portTICK_RATE_MS)
#define SECONDS(x) ((portTickType) (x * 1000) / portTICK_RATE_MS)

// create a non-reloading timer
xTimerHandle timerHandle = xTimerCreate((const char * const) "My Timer",
       SECONDS(12), pdFALSE, NULL, myTimerCallback);

// change the timer period from 12 to 8 seconds
xTimerChangePeriod(timerHandle, SECONDS(8), MILLISECONDS(10));

// start counting from 0 again
xTimerReset(timerHandle, MILLISECONDS(10));

// stop the timer
xTimerStop(timerHandle, MILLISECONDS(10));

// delete the timer
xTimerDelete(timerHandle, MILLISECONDS(10));
```

As soon as `xTimerStart` is called, the referenced timer starts counting up ticks until it reaches it's period. `xTimerReset` resets this count, so that the timer has to count up again from zero. The period of a timer can be adjusted with `xTimerChangePeriod`. This works even when the timer is already running. When the tick count becomes equal to the timers period, the timers callback function is executed. While it is running, a timer can be stopped with `xTimerStop`. This will reset the tick count and also prevent the timer from counting up again. The callback function won't be executed. A stopped timer can be started again at a later point with `xTimerStart`. If a timer (running or not) is not needed anymore, it can be deleted with `xTimerDelete`.

Keep in mind that there are special versions of these functions to be used from within interrupt service routines (ISRs). They usually have the suffix "FromISR", such as `xTimerStartFromISR()` or `xTimerStopFromISR()`.

Further information on the usage of queues can be found on the FreeRTOS homepage:
http://www.freertos.org/RTOS-software-timer.html

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2.5 Queues

Queues are the primary mechanism for intertask communication. The most common use case for queues is the producer-consumer-pattern. In this case, two tasks communicate via a queue. One task, called the producer, puts data into the queue as it becomes available. Another task, called the consumer, polls the queue for data and handles it. The advantage of the queues of the FreeRTOS API is the fact, that they are threadsafe. This means, that even if two or more tasks want to access the queue at the same time, only one task will put something into the queue or get something from the queue, while all the others are blocked and waiting for their turn.

As an example, this chapter will show the implementation of a producer-consumer example, where the producer and consumer are a task respectively, which communicate over a FreeRTOS-queue.

To use queues, the corresponding header-file must be included. This is shown in **Code 9**. Keep in mind that the header-file `queue.h` must be included after the header-file `FreeRTOS.h`.

Code 9. Including the API

```
// after #include "FreeRTOS.h"
#include "queue.h"
```

The first entity is the producer. The task's function is declared in **Code 10**. The task will get the queue as its input, which is why the input `pParameters` is first cast to a variable `myQueue` of type `QueueHandle_t`. The queue handle is used in most functions of the queue API.

The producer enters an infinite loop, in which he attempts to call `xQueueSend()` in every iteration. Its inputs are the previously mentioned queue handle, a pointer to the value that will be sent, and a timeout. In this implementation, the timeout is 100 ticks. If the timeout has ended, the value will not be sent to the queue, and the next loop iteration will start. If `xQueueSend()` was successful, i.e. the return value is `pdPASS`, then the task will print the value to the console.

Code 10. Producer Function

```
static void produce(void *pParameters) {
    QueueHandle_t myQueue = (QueueHandle_t) pParameters;
    int count = 0;
    BaseType_t result = pdFAIL;
    while (true) {
        result = xQueueSend(
                     myQueue,          // handle of the queue
                     (void *) &count,  // pointer to the value to send
                     100               // timeout
                 );
        if(pdPASS == result) {
            printf("Value produced: %d\n\r", count);
        }
        count++;
    }
}
```


© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

The second entity is the consumer. The task's function is declared in **Code 11**. Just like for the producer, the function's input is the queue and will be stored in the variable `myQueue` accordingly.

The function enters an infinite loop as well, but this task will call `xQueueReceive()` in every iteration. It has similar inputs to `xQueueSend()`. The first being the queue-handle, the second is the pointer to the variable where the received value will be stored in, and finally the timeout. In this implementation, the timeout is 1000 ticks, which is higher than the producer's timeout. Additionally, after every iteration, the task is blocked for 2000 additional ticks, for demonstration purposes. This allows the producer to give more values to the queue than the consumer can receive.

Code 11. Consumer Function

```
static void consume(void *pParameters) {
    int value = -1;
    BaseType_t result = pdFAIL;
    QueueHandle_t myQueue = (QueueHandle_t) pParameters;
    while (true) {
        result = xQueueReceive(
                     myQueue,   // handle of the queue
                     &value,    // pointer to store the value in
                     1000       // timeout
                  );
        if(pdPASS == result ) {
            printf("Value consumed: %d\n\r", value);
        }
        vTaskDelay(2000);
    }
}
```

Finally, the producer task, the consumer task and the queue have to be initialized. This is done in the following snippet **Code 12**. The handles to the tasks and to the queue should be global, so they can be accessed by other functions eventually. Inside `initQueues()`, the first step is initializing the queue. This is done using the function `xQueueCreate()`, which receives two inputs. The first input is the number of items, which the queue will be able to hold. The second input is the size of each item. The size is measured in bytes, which is why 4 is used here. Finally, if the queue was allocated successfully, the tasks will be started. The function from **Code 12** can be directly used in appInitSystem() to start the example implementation.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 12. Initializing the Queue and Tasks

```
static xTaskHandle producerTask = NULL;
static xTaskHandle consumerTask = NULL;
static QueueHandle_t queue = NULL;

/** use this function in appInitSystem to start the example **/

void initQueues(void) {
    // create a queue that can store up to 10 values of type uint32_t
    queue = xQueueCreate(
                  10,         // queue capacity
                  4 // size of individual int values (in bytes)
               );
    if(queue != NULL) {
        xTaskCreate(produce, "Producer", configMINIMAL_STACK_SIZE, queue,
                  tskIDLE_PRIORITY, &producerTask);
        xTaskCreate(consume, "Consumer", configMINIMAL_STACK_SIZE, queue,
                  tskIDLE_PRIORITY, &consumerTask);
    }
}
```

If the tasks and the queue are not needed anymore, the following function can be used to delete the tasks and the queue.

Code 13. Deleting Tasks and Queue

```
void deinitQueues(void) {
    vTaskDelete(producerTask);
    vTaskDelete(consumerTask);
    vQueueDelete(queue); // delete the queue after stopping all tasks
}
```

Note that there are special versions of the functions used for receiving from a queue and sending to a queue, to be used from within interrupt service routines (ISRs). They have the suffix `FromISR`. For example `xQueueSendFromISR()` or `xQueueReceiveFromISR()`

Further information on the usage of queues can be found on the FreeRTOS homepage:
http://www.freertos.org/Inter-Task-Communication.html

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

2.6 Semaphores

Semaphores are used to synchronize tasks and to restrict access to exclusive resources. The code that is restricted by a semaphore is called *critical section*. On implementation level, semaphores actually are special cases of queues that don't transfer data. The length of the underlying queue depends on the type of the semaphore. Binary semaphores, for example, are queues of length 1. Tasks can '**take**' them and '**give**' them back again. Once a semaphore was taken, others can't take that semaphore until it is given back again. This mechanism can, for example, be used to make one tasks wait for an event in a different task.

As an example, this chapter will show the implementation of a situation, where multiple tasks are competing for a resource (in this case, the resource is a counter). The resource is restricted using a binary semaphore.

To use semaphores, the corresponding header-file must be included. This is shown in **Code 14**. Keep in mind that the header-file `semphr.h` must be included after the header-file `FreeRTOS.h`.

Code 14. Including the API

```
// after #include "FreeRTOS.h"
#include "semphr.h"
```

Each competing task uses the same function, for trying to access the resource. The function is declared in **Code 15**. Each task will get the semaphore as its input, which is why the input `pParameters` is first cast to a variable `taskSemaphore` of type `SemaphoreHandle_t`. The semaphore handle is used in most functions of the semaphore API.

Each task enters an infinite loop, in which it attempts to call `xSemaphoreTake()` in every iteration. Its inputs are the previously mentioned semaphore handle and a timeout. In this implementation, the timeout is 60000 ticks. If the timeout has ended the semaphore will not be taken, and the next loop iteration will start. If `xSemaphoreTake()` was successful, i.e. the return value is `pdTRUE`, then the task will first print the current value of the counter, wait for 1000 ticks before incrementing the counter. Finally, the semaphore is given back using the function `xSemaphoreGive()`.

The delay between printing the counter and incrementing the counter is a means to demonstrate the effects of the semaphore. If the semaphore does not work properly and two tasks may access counter at the same time, the same value of `counter` might be printed twice, since the counter is only incremented after the delay.

Note: The variable `counter` is declared static in this function, which is why the variable will only be declared once during the entire runtime, even if `reserve()` is called multiple times.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 15. Reserver Function

```
static void reserve(void *pParameters) {
SemaphoreHandle_t taskSemaphore = (SemaphoreHandle_t) pParameters;
    BaseType_t result = pdFAIL;
    static int counter = 0;
    while (true) {
        result = xSemaphoreTake(taskSemaphore, 60000);
        if( pdTRUE == result )  {
            printf("counter: %d\n\r", counter);
            vTaskDelay(1000);
            counter++;
            xSemaphoreGive(taskSemaphore);
        }
        vTaskDelay(500);
    }
}
```

Finally, the reserver tasks and the semaphore have to be initialized. This is done in the following snippet **Code 12**. The handles of the tasks and of the semaphore should be global, so they can be accessed by other functions eventually. Inside `initSemaphores()`, the first step is initializing the semaphore. This is done using the function `xSemaphoreCreateBinary()`. Initially, the semaphore is in a closed state. This is why the function `xSemaphoreGive()` has to be used after creation, to ensure that a task may take the semaphore. If the semaphore was allocated and opened successfully, the tasks will be started. The function from **Code 16** can be directly used in `appInitSystem()` to start the example implementation.

Code 16. Initializing the Semaphore and Tasks

```
static xTaskHandle reserverTask1 = NULL;
static xTaskHandle reserverTask2 = NULL;
static xTaskHandle reserverTask3 = NULL;
static SemaphoreHandle_t semaphore = NULL;

void initSemaphores(void) {
      semaphore = xSemaphoreCreateBinary();
    if(semaphore != NULL) {
        BaseType_t result = xSemaphoreGive(semaphore);
        if( pdTRUE == result ) {
            // create three tasks that compete for the semaphore
            xTaskCreate(reserve, "Reserver1", configMINIMAL_STACK_SIZE,
            semaphore, tskIDLE_PRIORITY, &reserverTask1);
            xTaskCreate(reserve, "Reserver2", configMINIMAL_STACK_SIZE,
            semaphore, tskIDLE_PRIORITY, &reserverTask2);
            xTaskCreate(reserve, "Reserver3", configMINIMAL_STACK_SIZE,
            semaphore, tskIDLE_PRIORITY, &reserverTask3);
        }
    }
}
```

If the tasks and the queue are not needed anymore, the following function can be used to delete the tasks and the queue.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 17. Deleting Tasks and Semaphore

```
void deinitSemaphores(void) {
    vTaskDelete(reserverTask1);
    vTaskDelete(reserverTask2);
    vTaskDelete(reserverTask3);
    vSemaphoreDelete(semaphore); // delete the semaphore after stopping tasks
}
```

The example in Code 16 creates a binary semaphore using `xSemaphoreCreateBinary()`. A binary semaphore only allows one task to be inside the critical section at the same time. Constructors for other types of semaphores can be found in the API reference for the semaphore module here.

In this example, all the tasks do is take the semaphore, print a counter, and give it back. In a real application, the semaphore could, for example, be used to limit write access to a file. In this case, any task that wanted to write to that file would first call `xSemaphoreTake()`. If another task had already acquired the semaphore, this call would block until the other task would give the semaphore back. As soon as a task acquires the the semaphore, it is allowed to modify the file. When it is done, the task has to call `xSemaphoreGive()` so that other tasks get the chance to write to the file. This situation also leads to a common source of errors when working with semaphores: if one task fails to give back a semaphore while other tasks are blocked waiting for it, the application can run into a deadlock.

Note that there are special versions of the functions used for taking a semaphore and giving a semaphore, to be used from within interrupt service routines (ISRs). They have the suffix `FromISR`. For example `xSemaphoreTakeFromISR()` or `xSemaphoreGiveFromISR()`

Further information on the usage of semaphores can be found on the FreeRTOS homepage:
http://www.freertos.org/Inter-Task-Communication.html

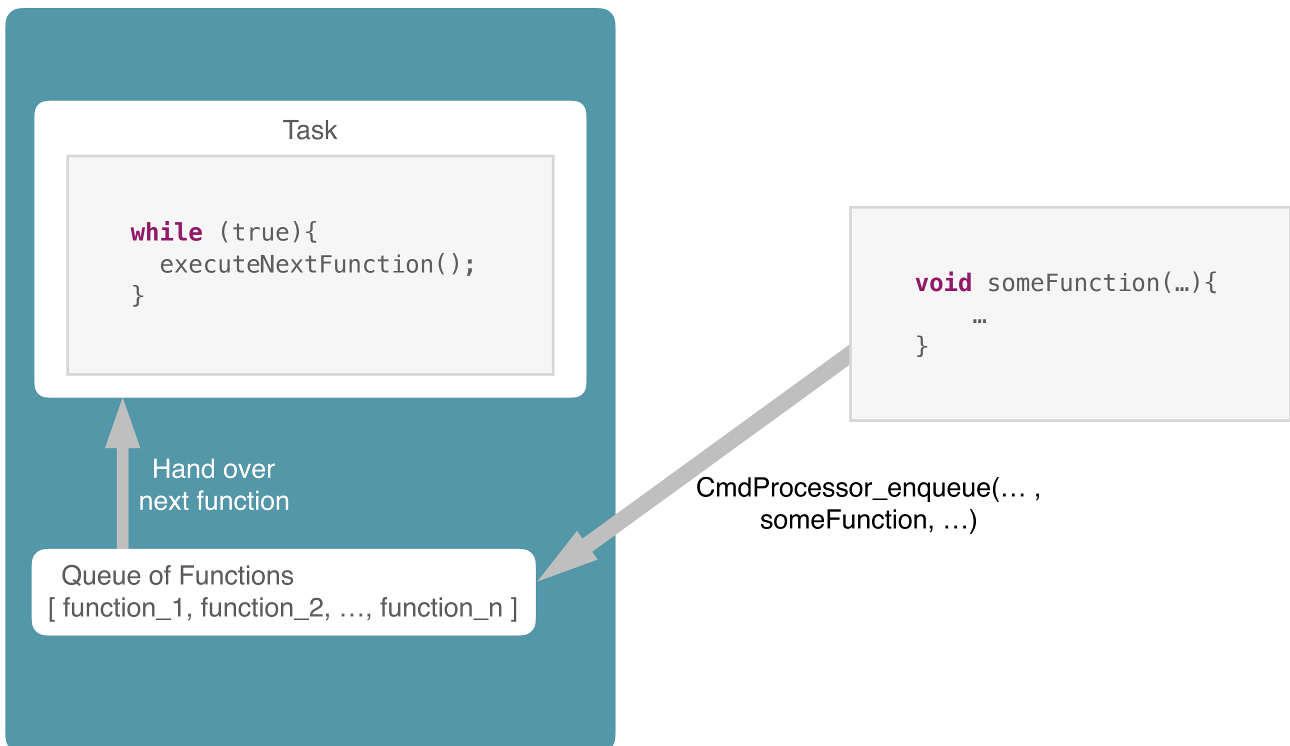
© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

3 CmdProcessor

With the release of Workbench 3.0, a new feature was added, which is called *CmdProcessor.* This is essentially a combination of the features *task* and *queue,* that lets the developer send functions to the CmdProcessor, which will then be processed by the internal task. This allows for more control of the environment in which a function will run, and provides a strict order of execution. Additionally, the function `appInitSystem()` is also called in the context of a CmdProcessor, and receives the handle for the CmdProcessor as its input. This will be later explained in this chapter.

Picture 3 shows the structure and the main interaction between application and CmdProcessor.

Picture 3. CmdProcessor structure



The CmdProcessor is provided by the header-file `BCDS_CmdProcessor.h`. To make it available in your code, the header-file must be included, as shown in **Code 11**.

Code 18. Including the API

```
#include "BCDS_CmdProcessor.h"
```

The essential functions and types it implements are listed in **Table 3**.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Table 3. CmdProcessor API excerpt

API	Description
CmdProcessor_T	This type holds the structure of the CmdProcessor. It has the internal task and queue as fields. Additionally, its name can be accessed. The reference to a CmdProcessor_T has to be passed to every function of the API.
CmdProcessor_initialize()	This function initializes the CmdProcessor. The signature is similar to xTaskCreate. This function receives a name, the task priority, the task stack depth, and the queue size as inputs.
CmdProcessor_enqueue()	This function adds a function to the queue of the CmdProcessor. As input, it receives the function, which will be added, and two input parameters for the function.

To use the API, a variable of type `CmdProcessor_T` has to be declared and initialized via `CmdProcessor_initialize()`. This is shown in **Code 12**. It has four inputs, the CmdProcessor itself, a name for the task, a priority, a stack depth and a queue size. The priority and stack depth should be chosen according to the nature of the functions this CmdProcessor has to process. In any case, the priority must be lower than the priority of the FreeRTOS scheduler. If the functions need much space, a higher stack depth should be chosen. The queue size determines how many functions can be queued, and thus be waiting for execution, at the same time.

It is recommended to make the CmdProcessor and the pointer a global variable. If the variables are deleted and their content overwritten, it might lead to unexpected behaviour.

Code 19. Initializing the CmdProcessor

```
static CmdProcessor_T myCmdProcessor; // this should be global
static CmdProcessor_T *myCmdProcessor_ptr = &myCmdProcessor; // same here

CmdProcessor_initialize(
    myCmdProcessor_ptr,         // pointer to the CmdProcessor_T
    "my cmd processor",         // the internal task's name
    tskIDLE_PRIORITY,           // the internal task's priority
    configMINIMAL_STACK_SIZE,   // stack depth (stack size is stack depth * 4)
    4                           // the internal queue's size
)
```

For a function to be enqueued, it has to have a specific signature. **Code 13** shows an example for such a function. It is static, has return-type void and the only inputs are two parameters, the first being a void pointer, the second one being of type `uint32_t`. These parameters will be given to the function at runtime by the function `CmdProcessor_enqueue()`.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 20. Example Function

```
#include "stdio.h"  // if not already included

static void myFunction(void *param1, uint32_t param2)
{
    (void) param1;
    (void) param2;
    printf("Function Processed!\r\n");
}
```

Finally, now that a function is written, it has to be enqueued. **Code 14** shows how to enqueue the function `myFunction` from **Code 13**. While this code only enqueues this function one time, it can be enqueued multiple times, and there is essentially no limit to how often a function can be enqueued. The only limit is the queue size. The queue size determines the number of functions that can be queued at the same time

Code 21. Enqueueing a Function

```
CmdProcessor_enqueue(
    myCmdProcessor_ptr,          // pointer to the CmdProcessor_T
    myFunction,                  // the function to be queued
    NULL,                        // parameter 1
    0                            // parameter 2
)
```

These code snippets will be enough to get by with the CmdProcessor for most applications, but another great advantage compared to a task is, that the CmdProcessor can call the same function with different parameters every time. While **Code 14** uses NULL and 0 for the parameters respectively, parameter 1 can be a pointer to a variable of any type and parameter 2 can be any unsigned 32-bit integer value. **Code 15** shows how to use these parameters. The first function, `myParameterFunction()` will be enqueued, and parameter 1 will point to an integer. Before the integer is used inside the function, it should be verified that the pointer does not point to NULL (i.e. nothing). Parameter 2 can be safely used. The second function will enqueue the first function with predetermined values.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 22. Enqueue with parameters

```
#include "stdio.h"  // if not already included

static void myParameterFunction(void *param1, uint32_t param2)
{
    if(param1 != NULL){
      int *myInteger = param1
      printf("My favorite integers are: %d and %d\n\r",
                *myInteger, (int) param2);
   }
   printf("Function Processed!\r\n");
}
```

This concludes how to generally use CmdProcessor, but as mentioned before, the entry point of every application `appInitSystem()` is called within the context of a CmdProcessor as well, and additionally, the first parameter of `appInitSystem()` is the pointer to the CmdProcessor, which can be consequently reused within the application. **Code 16** shows how to retrieve the CmdProcessor with a pointer. To be able to use this CmdProcessor outside of the function, make sure to create a global variable for the pointer, otherwise the pointer will be lost after `appInitSystem()` has been executed.

While manually initialized CmdProcessors had to be made global, the `appCmdProcessor` variable can stay local to the function `appInitSystem()`, because the `appCmdProcessor` has been initialized and made global outside of `appInitSystem()`.

Code 23. CmdProcessor from appInitSystem

```
CmdProcessor_T *appCmdProcessor;

static void appInitSystem(void CmdProcessorHandle, uint32_t param2)
{
    if(CmdProcessorHandle == NULL){
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    appCmdProcessor = (CmdProcessor_T *) CmdProcessorHandle;
    CmdProcessor_enqueue(myCmdProcessor_ptr, myFunction, NULL, 0)
}
```


© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

4 Memory Management

The most common problems associated with tasks are stack overflows. This occurs, when a task tries to allocate more data than it is allowed to. A stack overflow is an application-breaking issue, and it is considerarbly difficult to analyze why stack overflow is happening, especially in large projects. This chapter gives an introduction to how the RAM and the flash memory are used on the XDK, and especially the role of FreeRTOS and the allocation of tasks.

4.1 Flash Memory

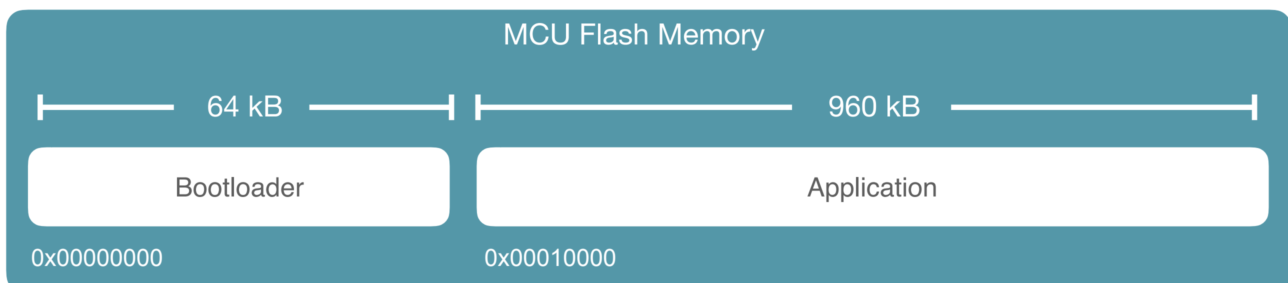
The XDK's microcontroller is equipped with 1MB (= 1024kB) of flash memory. Flash memory is not lost after restarting the XDK, which is why the flash memory is mainly used to store the bootloader and the application code.

The bootloader is the first piece of code that runs after starting the XDK. Its responsibility lies in setting the XDK into a proper state, before starting the actual application. The bootloader is write-protected, and is the only piece of software on the XDK that shall not be modified by the XDK user.

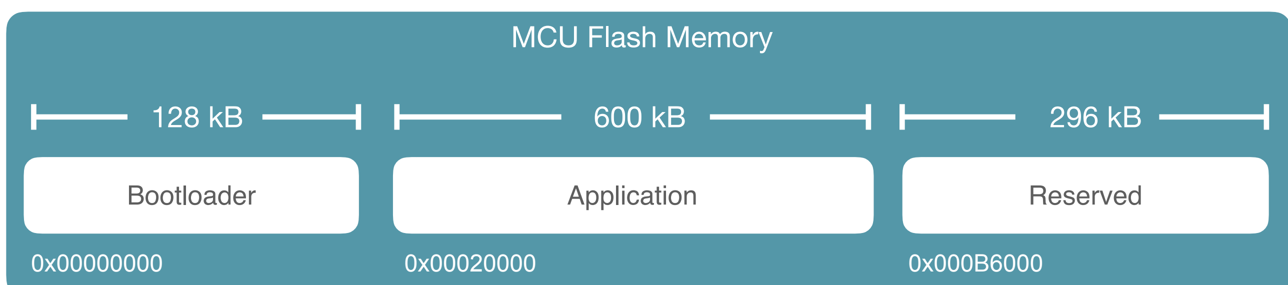
The application code is runnable code, which had been compiled and flashed onto the XDK's flash memory, using the XDK-Workbench.

Depending on the bootloader version, the flash memory is partitioned differently. **Picture 4** shows the bootloader flash memory partitioning for version 0.0.11 and below, and **Picture 5** shows the partitioning for versions 1.0.0 and 1.1.0.

Picture 4. Flash memory partioning for bootloader version 0.0.11



Picture 5. Flash memory partioning for bootloader versions 1.0.0 and 1.1.0



Note that in versions 1.0.0. and 1.1.0 the space marked as *reserved* will be used for the application, if the application is flashed using FOTA (=Firmware Over The Air). But, if the application is flashed over USB, bootloader and application use a total of 728kB of memory size, which leaves 296kB of memory which is essentially unused.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Consequently, the memory can be written on by the XDK user, which allows for data to be kept even after the XDK had been turned off.

There is API that allows writing onto the flash memory. It is made available by including the header-file `BCDS_MCU_Flash.h`, as shown in **Code 24**.

Code 24. Including the API

```
#include "BCDS_MCU_Flash.h"
```

The function in **Code 2,5** shows how to read four bytes from the flash memory, starting at the first location marked as *reserved*, which is 0x000B6000 in hexadecimal system, or 745472 in decimal system. The API's function `MCU_Flash_Read()` requires a pointer to the address. In this case, the address is stored in a variable of type `uint32_t`, holding the value 0x000B6000. Furthermore, a buffer and the length to read are used as input. For visualization, the four bytes are printed to the console.

Code 25. Reading from Flash Memory

```
void read(void) {
    uint32_t readAddress = 0x000B600;
    uint8_t buffer[4];
    uint32_t length = sizeof(buffer);
    MCU_Flash_Read((uint8_t *) readAddress, buffer, length);
    printf("Memory: %d, %d, %d, %d\n\r", buffer[0], buffer[1], buffer[2],
               buffer[3]);
}
```

Writing is done quite similarly. **Code 26** shows how to write four bytes to the flash memory, starting at the first location marked as *reserved* which is 0x000B6000 in hexadecimal system, or 745472 in decimal system. The API's function `MCU_Flash_Write()` requires a pointer to the address. In this case, the address is stored in a variable of type `uint32_t`, holding the value 0x000B6000. Furthermore, a buffer and the length to read are used as input. For visualization, the four bytes are read from the flash memory afterwards and printed to the console.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

Code 26. Writing to Flash Memory

```
void write(void) {
   uint32_t writeAddress = 745472;
   uint8_t buffer[4] = {1,2,4,8};
   uint32_t length = sizeof(buffer);
   MCU_Flash_Write((uint8_t *) writeAddress, buffer, length);
   memset(buffer, 0, 4);
   MCU_Flash_Read((uint8_t *) writeAddress, buffer, length);
   printf("Memory: %d, %d, %d, %d\n\r", buffer[0], buffer[1], buffer[2],
            buffer[3]);
}
```

There are a few things to keep in mind, when writing to the flash memory. The first being that the application code inside the flash memory can actually be overwritten, because only the bootloader is write-protected. Overwriting application code will lead to unexpected behaviour and should be avoided at all cost. Even writing to the bootloader segment should be avoided.

Second, although the flash memory is persistent through restarting the XDK, it will not persist through flashing a new application. When a new application is flashed, every unused bit of the flash memory will be set to 1. Since the flashing-process does not use the reserved segment, the reserved segment will always be cleared.

4.2 Random Access Memory (RAM)

The XDK is equipped with 128kB of RAM, which is used by the application at runtime. It is mainly comprised of the program stack, used and managed by the MCU, and the heap, used and managed by FreeRTOS. The stack has its base address at the last address of the RAM, while the heap has its base at zero. In a sense, the stack is growing downwards and the heap is growing upwards.

The key difference between the stack and the heap is that the heap is actually fixed in size, while the stack can grow dynamically. That means, allocating anything on the heap that exceeds the remaining available space of the heap is not possible. In contrast, if the stack grows too large, the stack and the heap may still collide.

The heap can be configured in the header-file `FreeRTOSConfig.h` located in the SDK at:

xdk110 > Common > config

The original values should not be overwritten without full knowledge of their implications. For more information, please refer to the FreeRTOS Configuration page here.

Inside the header-file in line 159 `configTOTAL_HEAP_SIZE` is defined as 61kB. This heap-size accommodates tasks, queues, semaphores and other FreeRTOS entities, which are allocated at runtime. The FreeRTOS heap is actually preceeded by the FreeRTOS kernel, which is the core of the operating system, responsible for allocation of memory on the heap, managing tasks, etc. The FreeRTOS kernel has a fixed size, independent of the configured heap-size. If the total heap size is set too high, there will not be enough space for the stack on the RAM, and the application may crash because of a heap and stack collision after a short amount of runtime.

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

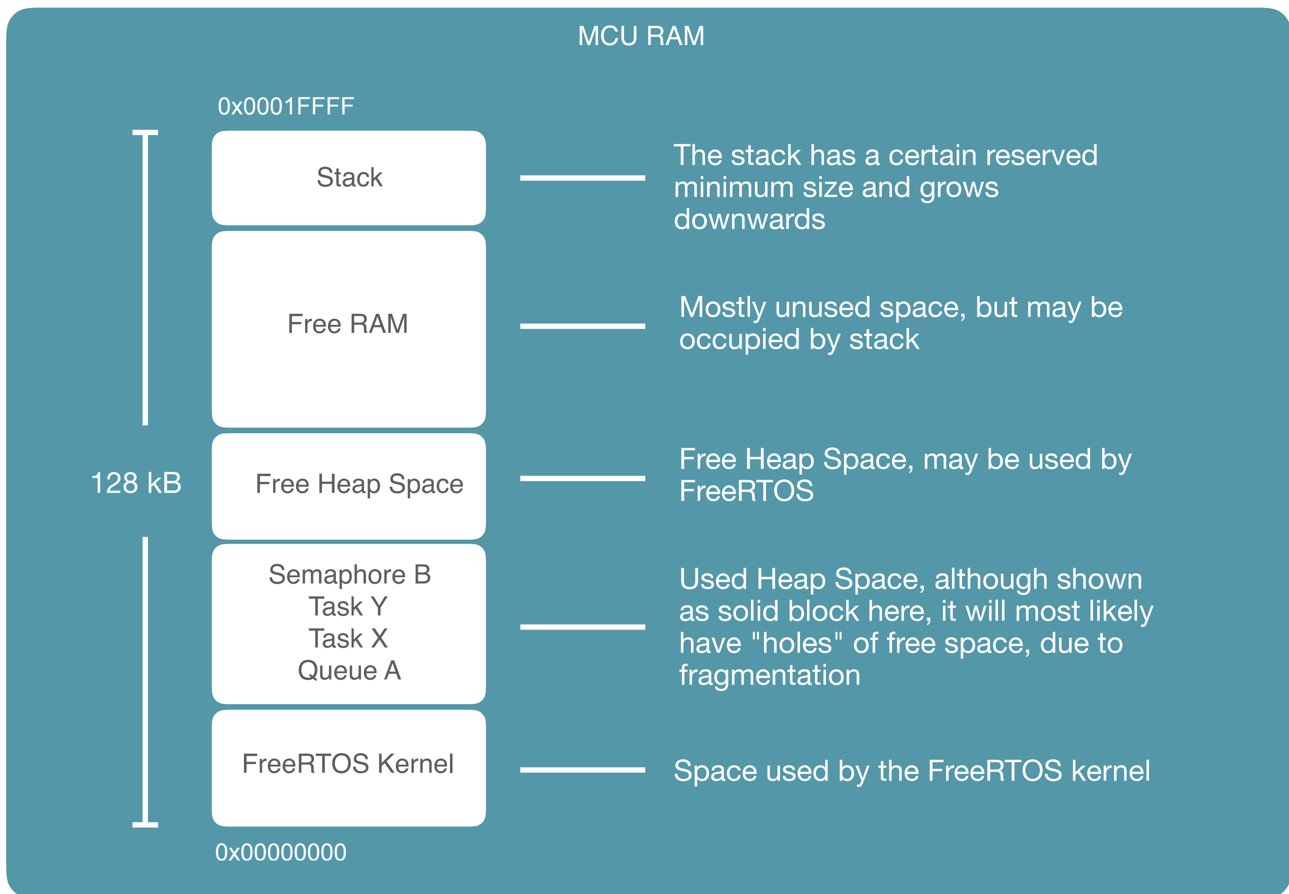
Tasks each have their respective stack. This stack is limited in size, and a task will only be initialized if the heap offers, enough space for the chosen stack-size. Otherwise, the function `xTaskCreate()` will fail. Every task has its entry function, that is used as an input in `xTaskCreate()`, and every function that is called from within a task (including function calls within the entry function), requires space on the stack. Space is required even if no variables are used within a function. That means recursive function only calling itself over and over again will eventually lead to a stack overflow.

Note: Allocating an array that requires more bytes than the stack can hold in total will also lead to a stack overflow.

The FreeRTOS kernel will prevent any operation on the heap from overflowing, which is why tasks are the safest way to design an application on the XDK.

Note: The function `appInitSystem()`, which is the entry point of every application, is called within the context of a task.

Picture 6. Visualization of the RAM usage



,

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.

5 Document History and Modification

Rev. No.	Chapter	Description of modification/changes	Editor	Date
2.0		Version 2.0 initial release	AFS	2017-08-17
2.1	2.4,2.5, 4	New queue and semaphore examples, Added Memory Management section	AFS	2017-10-05

© Bosch Connected Devices and Solutions GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties. BOSCH and the symbol are registered trademarks of Robert Bosch GmbH, Germany.
Note: Specifications within this document are subject to change without notice.