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	1.2
Document release date	06/03/2017
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 explain how to use common FreeRTOS features such as tasks, timers or semaphores. For further documentation and code examples, please refer to www.freertos.org.

Table of Contents

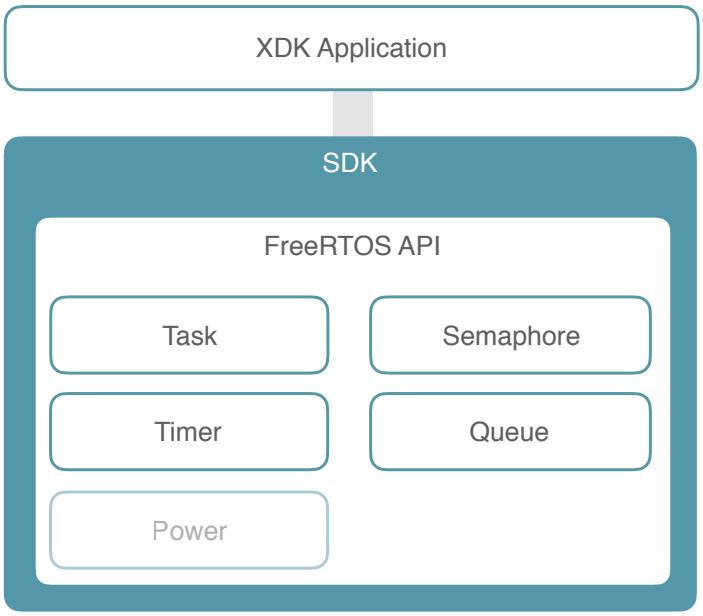
This guide postulates a basic understanding of the XDK and according Workspace. For new users we recommend going through the following guides at xdk.io/guides first:

- *Workbench Installation*
- *Workbench First Steps*

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 defines a number of functions and data types that are related to a specific kind of system-provided data structure or resource. The following picture gives an overview of the different software components.

Picture 1. Software Architecture



Each module can be linked into an application by including the corresponding system header file. The only exclusion is the power management module, which can only be accessed via a wrapping header called `BCDS_PowerMgmt.h` and which won't be covered in this guide.

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.

2. API Usage

This chapter will demonstrate how to use the most important modules of the FreeRTOS API.

2.1 Preparation

The XDK-Workbench offers a sample project that is called `XdkApplicationTemplate`. It can be opened 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 point at which the system hands over control 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

```
void appInitSystem(xTimerHandle xTimer)
{
    (void) (xTimer);

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

This entry point will be used to initialize and start any application. An application that only needs to execute some commands one after another could be completely implemented from within `appInitSystem()`. 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). Common means to achieve this behaviour are 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 XDK and the PC that's connected to it via USB, text that is printed to the console in `appInitSystem()` won't be visible in the Workbench console.

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 two header files: `FreeRTOSConfig.h` and `projdefs.h`. Both files are part of the SDK and themselves included in a top-level header called `FreeRTOS.h`. So by including that header, the application gains access to all required constants and definitions:

Code 2. Including the FreeRTOS header

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

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

```
void doSomething(void)
{
    xTaskHandle taskHandle = NULL;

    BaseType_t taskResult = 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 task handle for late reference
     );

     if(pdPASS != taskResult) {
         assert(pdFAIL);
     }
}
```

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. Also make sure that `appInitSystem()` always is the last function in the source file.

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

Table 1. Arguments of xTaskCreate

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	A string constant that will be used as the name of the new task. This name will only be used internally or for debugging purposes. Any string could be chosen.
Stack Depth	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` (lowest) and `configMAX_PRIORITIES` - 1 (highest). The priority `configMAX_PRIORITIES` (5) itself is reserved for interrupt handling.
Created Task	A pointer to a `xTaskHandle` that will store the created task object.

The return code of `xTaskCreate()` indicates whether the task was created successfully. If there wasn't enough memory left to create a stack with the requested size, it will be equal to `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`. If the return code is equal to `pdPASS`, the task was successfully created.

The next snippet gives an example for a task function.

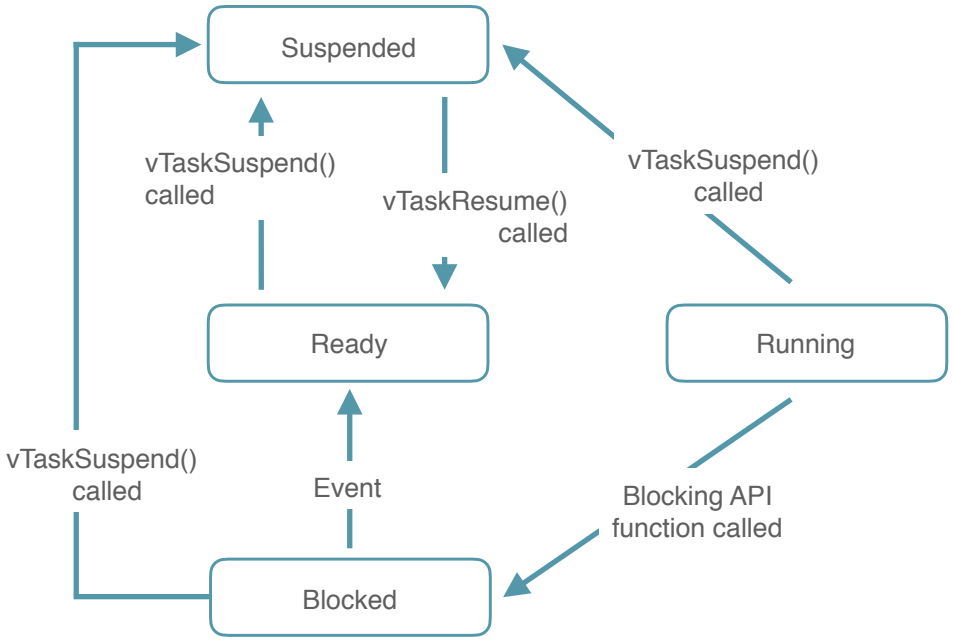
Note: The task function has to be declared before it can be used in `xTaskCreate()`. This can be achieved by placing it above the code that creates the task (**Code 3**).

Code 4. Implementing a task

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

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

    for (;;) {
       vTaskDelay(SECONDS(3)); // suspend task for 3 seconds

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

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:

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 RUNNING
// ...
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!

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 // static callback function
     );

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

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

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

`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 that is called when the timer expires. This 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.

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()`.

Notice that there are special versions of these functions to be used from within interrupt service routines (ISRs).

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

2.5 Queues

Queues are the primary form of intertask communication. They can be used to send data between tasks: One task, called the producer, puts data into a queue as soon as it becomes available (e.g. a network stack). Another task, called the consumer, polls the queue for data and handles it (e.g. a packet handling thread). While the queue is empty, the consumer task is blocked. As soon as data becomes available, the consumer starts processing it.

The following code snippet demonstrates how to create and use a queue:

Code 9. Queues

```
#include "queue.h" // include the queue module

#define MILLISECONDS(x) ((portTickType) x / portTICK_RATE_MS)

void doSomething(void)
{
    // create a queue that can store up to 10 items of type uint32_t
    QueueHandle_t queueHandle = xQueueCreate( 10, sizeof( uint32_t ) );
    if(queueHandle == NULL) {
        assert(false);
        return;
    }

    // put the value 42 into the queue (block up to 10 milliseconds)
    uint32_t someValue = 42;
    BaseType_t queueResult = xQueueSend( queueHandle, ( void * ) &someValue,
                                                      MILLISECONDS(10));
    if(pdPASS != queueResult) {
        assert(false);
        return;
    }

    // poll the first value from the queue (block up to 10 milliseconds)
    uint32_t someOtherValue = 0;
    queueResult = xQueueReceive( queueHandle, &someOtherValue,
    MILLISECONDS(10));
    if(pdPASS != queueResult) {
        assert(false);
        return;
    }

    // at this point, someOtherValue is 42

    vQueueDelete(queueHandle);
}
```

Code 9 creates a queue that can store up to 10 unsigned 32-bit-integers (or any other value that is 32 bits wide). After successfully creating the queue, the value `42` is put into it by calling `xQueueSend()`. The last parameter defines a time interval after which the call will be cancelled if it hasn't succeeded yet. If some other task has called `xQueueReceive()` on this queue before, it would now become active again. In this case, however, `xQueueReceive()` is called immediately afterward on the same task, retrieving the value which was stored into the queue just a few lines

before. Usually `xQueueSend()` and `xQueueReceive()` are called in different tasks, as described earlier.

Notice that there are special versions of these functions to be used from within interrupt service routines (ISRs).

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

2.6 Semaphores

Semaphores are used to synchronize tasks and to restrict access to exclusive resources. From an implementation perspective, they are special cases of queues that don't actually transfer data. The length of the underlying queue depends on the type of the semaphore. Binary semaphores, for example, are queues of length 1. They can be '**taken**' and later '**given**' back by tasks. 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.

The following code snippet demonstrates how to create and use a semaphore:

Code 10. Semaphores

```
#include "semphr.h" // include the semaphore module

#define MILLISECONDS(x) ((portTickType) x / portTICK_RATE_MS)

void doSomething(void)
{
    // create a binary semaphore in the taken state
    SemaphoreHandle_t semaphore = xSemaphoreCreateBinary();
    if(semaphore == NULL) {
        assert(false);
        return;
    }

    // give back the initially taken semaphore
    BaseType_t semaphoreResult = xSemaphoreGive(semaphore);
    if(pdPASS != semaphoreResult) {
        assert(false);
        return;
    }

    // take the semaphore again (block up to 10 milliseconds)
    semaphoreResult = xSemaphoreTake( semaphore, MILLISECONDS(10) );
    if(pdPASS != semaphoreResult) {
        assert(false);
        return;
    }

    vSemaphoreDelete(semaphore);

}
```

In **Code 10**, `xSemaphoreCreateBinary()` is used to create a new binary semaphore. Constructors for other types of semaphores can be found in the API reference for the semaphore module: http://xdk.bosch-connectivity.com/xdk_docs/html/semphr_8h.html. The lines afterwards demonstrate how to give an take a binary semaphore. Note that `xSemaphoreCreateBinary()` creates a semaphore that is initially taken. The newly created 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 the the semaphore is acquired by a task, 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 example also shows 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 will run into a deadlock.

Notice that there are special versions of these functions to be used from within interrupt service routines (ISRs).

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