

ESW Examination assignment

Student name: Eduard Fischer-Szava

Student number: 283624

Topic: FreeRTOS and Test-Driven Development

Contents

Table of contents	2
Solution design.....	3
Solution structure	7
Application execution screenshots	8
Testing.....	9
Significant findings and observations	13
Implementation	14

Table of contents

Figure 1 - initialization module	3
Figure 2 - Application controller task module	3
Figure 3 - Configuration file (interface)	4
Figure 4 – Transmitter task module	5
Figure 5 - Water level task module	5
Figure 6 - Water Temperature task module	6
Figure 7 - Solution structure	7
Figure 8 - FreeRTOS_ESW_Exam configuration (Production code)	7
Figure 9 - FreeRTOS_ESW_EXAM_Tests configuration (Test project for the production code)	7
Figure 10 - Application execution: initial run	8
Figure 11 - Application execution: temperature value reset (cooldown)	8
Figure 12 - Application execution: transmission of the payload	8
Figure 13 - Experiment to showcase the understanding of the payload structure	9
Figure 14 - FFF setup	9
Figure 15 - Water_temperature_sensor_ADT_Test class	10
Figure 16 - WaterTemperatureSensor ADT unit tests	10
Figure 17 - Message Payload test class	11
Figure 18 - Message Payload test suite	12
Figure 19 - Test results	12
Figure 20 - waterTemperatureSensor_ADT.h (Water temperature sensor module interface)	13
Figure 21 - payload.h (Transmitter module interface)	13
Figure 22 - Dependencies (configuration.h)	13
Figure 23 - Main module	14
Figure 24 - Application Controller Module Interface	14
Figure 25 - Application Controller Module - part 1 (appControllerTask.c)	15
Figure 26 - Application Controller module - part 2 (appControllerTask.c)	15
Figure 27 - Application Controller module - part 3 (appControllerTask.c)	16
Figure 28 - Water Level Sensor Module interface (waterLevelSensor.h)	17
Figure 29 - Water Level Sensor Module part 1	17
Figure 30 - Water Level Sensor Module part 2	18
Figure 31 - Water Level Sensor Module part 3	19
Figure 32 - Water Temperature Sensor Module interface (waterTemperatureSensor.h)	20
Figure 33 - Water Temperature Sensor Module part 1	20
Figure 34 - Water Temperature Sensor Module part 2	21
Figure 35 - Transmitter module interface (transmitterTask.h)	22
Figure 36 - Transmitter Module part 1	23
Figure 37 - Transmitter module part 2	24

Solution design

For the complete design of the project, please see “Design class diagram FreeRTOS_ESW_Exam.svg”.

1. Initialization module

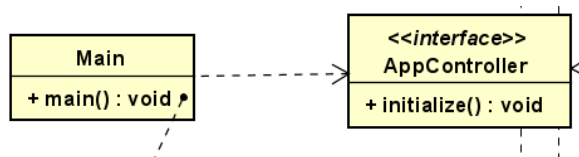


Figure 1 - initialization module

In the above diagram section (Figure 1), the initialization module is shown, the purpose of this module is within the scope of the main function to initialize the application control module (create the app controller). The relationship between the two modules is a loosely coupled dependency established using an interface that exposes to the application initialization module (Main) the initialize function to allow the external creation of the application control module (App Controller).

2. Control task module

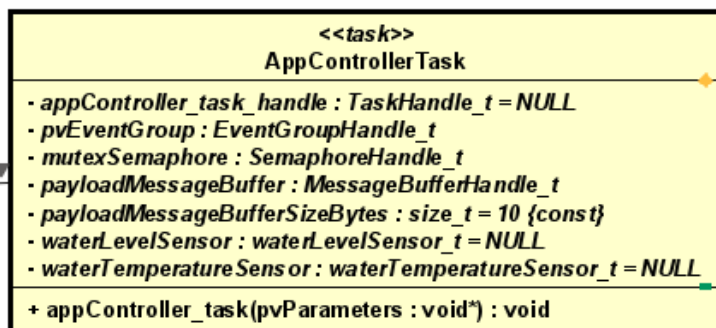


Figure 2 - Application controller task module

The Control Task Module (AppControllerTask, Figure 2), is the central module of the application, private (static) variables are represented, these will be later used for different application

functionalities such as the creation of the message buffer to which the payload ¹ will be sent (establishment of synchronization between the application controller module and the transmitter module), the mutex semaphore necessary for protection of print statements (resources), the two measurement sensor objects (ADTs) and the event group that allows synchronization with the water level sensor object to both initialize a measurement (will occur with a delay of ca. 100 ms) and retrieve the result of a measurement for the water level once it is completed .

Furthermore, the variable `appController_task_handle` is the task control block of the application controller task module and the function `appController_task(pvParameters : void*2) : void` represents the task function definition and later declaration (implementation of the task critical section).

Moreover, the application controller module interface has dependencies to all the other module interfaces (WaterTemperatureSensor, Transmitter, WaterLevelSensor) for the possibility of creating (initializing) these modules and interacting ³ with them (see “Design class diagram FreeRTOS_ESW_Exam.svg”).

3. Configuration file (interface)

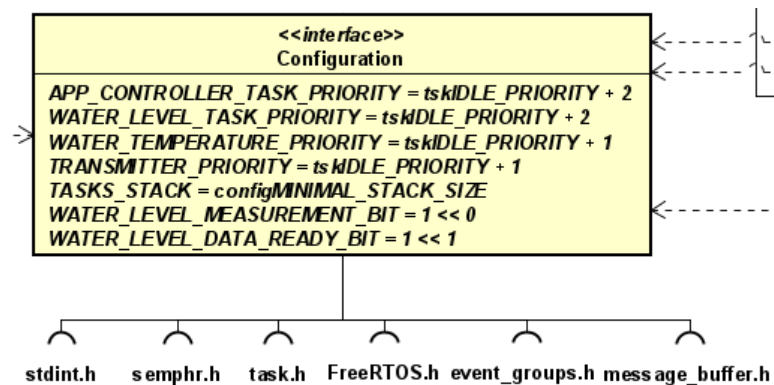


Figure 3 - Configuration file (interface)

Figure 3 showcases the use of an interface (header file) to contain the necessary parameters for the creation of the tasks within the measurement modules, the two bits that will be used for the synchronization between the application control module (AppControllerTask) and the water level measurement module (WaterLevelTask). As shown below the module itself, several external dependencies exist, mainly with the specific FreeRTOS external libraries that are mandatory for the creation of the tasks, the mutex, the event group and message buffer.

4. Transmitter task module

¹ Struct variable consisting of a byte array and its length used for transmission of the water level and temperature (measurements) through the message buffer to the transmitter module.

² `pvParameters : void*` - pointer variable parameters of type void pointer (point to anything)

³ In the case of the WaterTemperatureSensor and WaterLevelSensor aside from creating the modules, retrieval of measurement values.

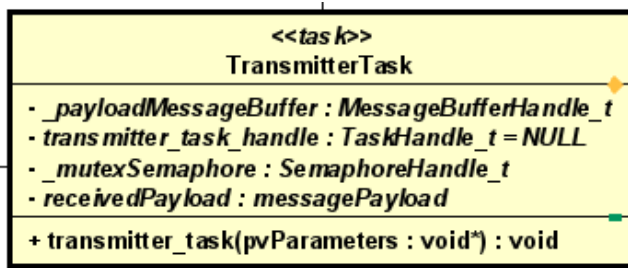


Figure 4 – Transmitter task module

The transmitter task module (Figure 4) consists of several private variables. The `_payloadMessageBuffer` is needed for the creation of the message buffer to which the application control module will be sending the payload once the measurements (of the measurement modules) have occurred. The `transmitter_task_handle` is the task control block for the task which will be used within the module. Mutex semaphore (`_mutexSemaphore`) is the variable that stores the mutex that will be passed to the transmitter module upon creation in the application control module for protection of the print statements (resources). Received payload (`receivedPayload`) is the variable in which the payload, once sent from the application control module, will be stored, and later having its contents displayed to the console. The function `transmitter_task(pvParameters: void*) : void` is the function that declares and later defines (the implementation of the function containing the task critical section) the transmitter task.

5. Water level task module

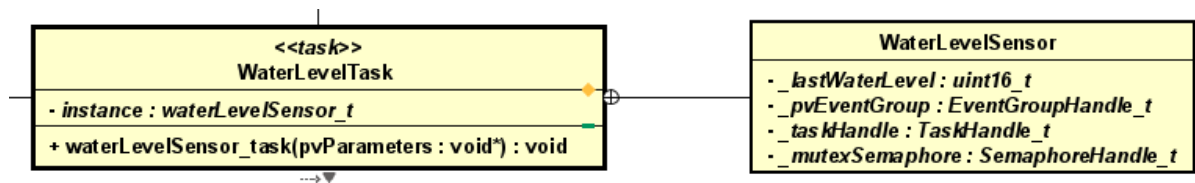


Figure 5 - Water level task module

Figure 5 displays the contents of the water level task module (WaterLevelTask and WaterLevelSensor ADT). The water level task module is composed of two parts, the water level task (WaterLevelTask, the task that will execute the measurement of the water level) and water level sensor (WaterLevelSensor, ADT object). The member variables of the WaterLevelSensor are `_lastWaterLevel` – used to store the latest measurement of the water level, `_pvEventGroup` – the event group variable that is used to store the event group parameter that will be sent to the WaterLevelTask by the application control module (AppController) for enabling the synchronization between the two tasks. The variable `_taskHandle` is used for the task creation (the task control block). The `_mutexSemaphore` variable has the purpose of protecting the print resources within the WaterLevelTask, the mutex is passed to the WaterLevelTask by the application control module upon task creation (from the AppController).

The variable “instance” is used as a unique identifier for the WaterLevelSensor object. Lastly, the `waterLevelSensor_task(pvParameters : void*) : void` is the function that declares and later defines the WaterLevelTask task function.

6. Water temperature task module

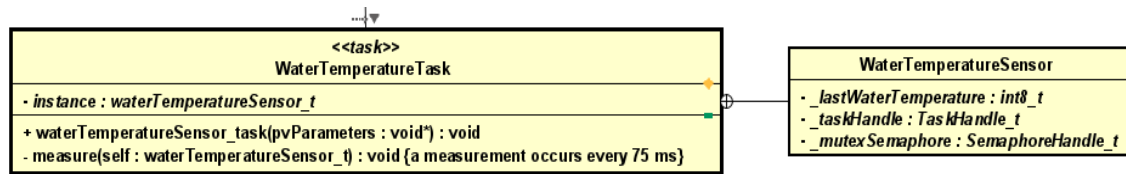


Figure 6 - Water Temperature task module

Figure 6 displays the contents of the water temperature task module (WaterTemperatureTask and WaterTemperatureSensor ADT), the composition of the module is similar to that of the WaterLevelTask module, the main difference is that the WaterTemperatureTask will execute measurements continuously⁴. The water temperature task module is composed of two parts, the water temperature task (WaterTemperatureTask, the task that will execute the measurement of the water temperature) and water temperature sensor (WaterTemperatureSensor, ADT object). The member variables of the WaterTemperatureSensor are `_lastWaterTemperature` – used to store the latest measurement of the water temperature. The variable `_taskHandle` is used for the task creation (the task control block). The `_mutexSemaphore` variable has the purpose of protecting the print resources within the WaterTemperatureTask, the mutex is passed to the WaterTemperatureTask by the application control module upon task creation (from the AppController).

The variable “instance” is used as a unique identifier for the WaterTemperatureSensor object. Lastly, the `waterTemperatureSensor_task(pvParameters : void*) : void` is the function that declares and later defines the WaterLevelTask task function.

⁴ With a delay of ca. 75 milliseconds.

Solution structure

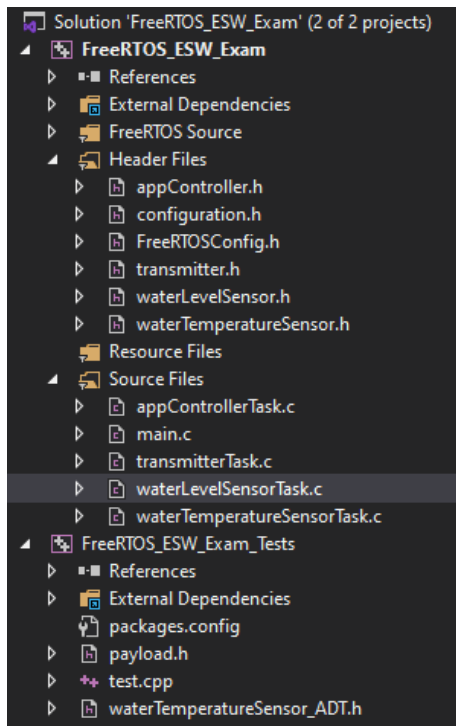


Figure 7 represents the structure of the solution, it is built upon two projects, FreeRTOS_ESW_Exam (production project and associated code) and FreeRTOS_ESW_Exam_Tests (the test project and associated code). In the beginning of the solution development, the structure was created alongside the necessary configuration(dependencies) which can be seen in Figure 8 and Figure 9.

Figure 7 - Solution structure

Include Directories

```
D:\FreeRTOS\FreeRTOSv202011.00\FreeRTOS\Source\portable\MSVC-MingW
D:\FreeRTOS\FreeRTOSv202011.00\FreeRTOS\Source\include
$(ProjectDir)
```

Figure 8 - FreeRTOS_ESW_Exam configuration (Production code)

Include Directories

```
D:\ESW_Exam\FreeRTOS_ESW_Exam\fff-master
D:\ESW_Exam\FreeRTOS_ESW_Exam\FreeRTOS_ESW_Exam
```

Figure 9 - FreeRTOS_ESW_EXAM_Tests configuration (Test project for the production code)

Application execution screenshots

Select D:\ESW_Exam\FreeRTOS_ESW_Exam\Debug\FreeRTOS_ESW_Exam.exe

```
Water level sensor task is running
Water temperature sensor task is running
Water temperature --- current: 31°
Transmitter started !

Water temperature --- current: 32°
Water level --- current (Liters): 101

Payload sent to the transmitter !

Payload was received --- content:

Byte[0] -- Water level -- : 0x0
Byte[1] -- Water level -- : 0x65
Byte[2] -- Water temperature -- : 0x0
Byte[3] -- Water temperature -- : 0x20
```

Figure 10 - Application execution: initial run

In Figure 10, the initial execution of the application is displayed. The first print statement to the console prints “Water level sensor task is running” symbolizing that the task (WaterLevelTask) was created, the empty line beneath is for separation of the prints to increase readability. The second print statement is “Water temperature sensor task is running” symbolizing once again that the second measurement task (WaterTemperatureTask) was created. The 3rd print statement displays the current value of the water temperature (31, initially set 30 on task creation). The 4th print statement symbolizes the creation of the transmitter task.

```
Water temperature --- current: 49°
Water temperature --- current: 50°
Water temperature exceeds 50° --- cooldown initiated

Water temperature --- current: 1°
Water temperature --- current: 2°
Water temperature --- current: 3°
Water temperature --- current: 4°
Water temperature --- current: 5°
```

Figure 11 - Application execution: temperature value reset (cooldown)

Following are the print statements that showcase the current values of the measurements (the water level starts with a hardcoded value of 100 that is incremented continuously by 1, the water temperature starts with a value of 30, once again, hardcoded and incremented by 1, however when reaching a value greater than 50, it will be assigned the value 1 - Figure 11).

```
Water temperature --- current: 5°
Water level --- current (Liters): 106

Payload sent to the transmitter !

Payload was received --- content:

Byte[0] -- Water level -- : 0x0
Byte[1] -- Water level -- : 0x6A
Byte[2] -- Water temperature -- : 0x0
Byte[3] -- Water temperature -- : 0x5
```

Figure 12 - Application execution: transmission of the payload

Figure 12 showcases the transmission from the AppControllerTask – “Payload sent to the transmitter” and retrieval at the Transmitter task of the payload – “Payload was received --- content: ”, the 4 following print statements display the hexadecimal contents of the received bytes (the last measurement of the water level and temperature). As it can be seen, the water temperature was 5° resulting in 0x0 ($16^3 * 0 + 16^2 * 0 = 0$) for the first byte – high byte and 0x5 for the second byte – low byte ($16^1 * 0 + 16^0 * 5 = 5$) or 0b0000 0000 0000 0101. The same concept is valid for the water level which has only its low byte containing a value (106 decimal, equivalent to $0x6A = 6 * 16^1 + 10 * 16^0 = 96 + 10$).


```

Water level --- current (Liters): 10002
Payload sent to the transmitter !
Payload was received --- content:
Byte[0] -- Water level -- : 0x27
Byte[1] -- Water level -- : 0x12
Byte[2] -- Water temperature -- : 0x0
Byte[3] -- Water temperature -- : 0x2E

```

Figure 13 - Experiment to showcase the understanding of the payload structure

Figure 13 showcases the result after changing the value of the `_lastWaterLevel` variable in the `WaterLevelSensorTask` to 10000 as initial value, here (at the capture moment - snippet, the value was incremented twice), Byte[0] – high byte is $0x27 = 2 * 16^3 + 7 * 16^2 = 8,192 + 1,792 = 9,984$ and Byte[1] – low byte is $1 * 16^1 + 2 * 16^0 = 16 + 2 = 18$.

Testing

Testing was conducted by using FFF⁵.

```

#include "gtest/gtest.h"
#include "../fff-master/fff.h"
#define_FFF_GLOBALS // Initialize the FFF framework

// External C-code files (necessary for testing)
extern "C"
{
#include "waterTemperatureSensor_ADT.h"
#include "payload.h"
}

```

Figure 14 - FFF setup

In Figure 14, the setup of the testing module is shown. The first dependency is the default google test project import, followed by the second dependency, that of the relative path to the location of the header file necessary for creating mocks (fakes). After initializing the framework, the `"waterTemperatureSensor_ADT.h"` and `"payload.h"` files are included as external C files for usage in the test suites.

⁵ Fake Function Framework

```

/*
 * Water temperature sensor ADT Test suite
 */

// Fake function to create a Water temperature sensor ADT object
FAKE_VALUE_FUNC(waterTemperatureSensor_t, waterTemperatureSensor_create);

// Fake function to retrieve the latest value of a measurement from a Water temperature sensor ADT object
FAKE_VALUE_FUNC(int8_t, waterTemperatureSensor_getLastWaterTemperature, waterTemperatureSensor_t);

class Water_temperature_sensor_ADT_Test : public ::testing::Test
{
protected:
    void SetUp() override
    {
        RESET_FAKE(waterTemperatureSensor_create);
        RESET_FAKE(waterTemperatureSensor_getLastWaterTemperature);
        FFF_RESET_HISTORY();
    }

    void TearDown() override
    {
    }
};

```

Figure 15 - Water_temperature_sensor_ADT_Test class

In Figure 15, the test class for the Water_temperature_sensor_ADT is shown, two function are being mocked, waterTemperatureSensor_create that mocks the creation of the water temperature sensor object (hence the first parameter of the fake function) and waterTemperatureSensor_getLastWaterTemperature that mocks the retrieval of the latest temperature after execution of a measurement (int8_t as the return type and waterTemperatureSensor_t as the argument of the function). The test class' SetUp() function will reset both function after execution of any test in which the functions are called, lastly, the entire call history will also be reset.

```

TEST_F(Water_temperature_sensor_ADT_Test, Test_is_temperature_sensor_ADT_create_called)
{
    // Arrange
    waterTemperatureSensor_t waterTemperatureSensor;
    // Act
    waterTemperatureSensor = waterTemperatureSensor_create();
    // Assert
    ASSERT_TRUE(waterTemperatureSensor_create_fake.call_count, 1);
    EXPECT_TRUE(waterTemperatureSensor == NULL);
}

TEST_F(Water_temperature_sensor_ADT_Test, Test_is_temperature_sensor_ADT_returning_a_value)
{
    // Arrange
    int8_t value;
    waterTemperatureSensor_t waterTemperatureSensor;
    // Act
    waterTemperatureSensor = waterTemperatureSensor_create();
    value = waterTemperatureSensor_getLastWaterTemperature(waterTemperatureSensor);
    // Assert
    ASSERT_TRUE(waterTemperatureSensor_create_fake.call_count, 1);
    ASSERT_TRUE(waterTemperatureSensor_getLastWaterTemperature_fake.call_count, 1);
    EXPECT_TRUE(waterTemperatureSensor_getLastWaterTemperature_fake.return_val == 0);
    EXPECT_EQ(value, 0);
}

```

Figure 16 - WaterTemperatureSensor ADT unit tests

In Figure 16, two tests are made. The first unit test is mocking the creation of the waterTemperatureSensor ADT object. We declare a variable of the type waterTemperatureSensor_t, waterTemperatureSensor, then we assign to it the function call of waterTemperatureSensor_create() which will return the an object of the same type, thus the variable will be initialized. Assertions are made to verify that the mock of the create() function (the fake function) was called exactly once and as the interface tested (waterTemperatureSensor_ADT.h) has no implementation, the variable's contents – waterTemperatureSensor are NULL. The second test mocks the retrieval of a measurement (the last water temperature). In the Arrange subsection, a variable to store the returned value of the measurement is declared – value of the same type as the return type of the getLastWaterTemperature() function. Within the Act subsection, the waterTemperatureSensor object is assigned the waterTemperatureSensor_create() function which will return a waterTemperatureSensor_t object instance, thus initializing the variable. Following is the assignment of the waterTemperatureSensor_getLastWaterTemperature() function that has as an argument a waterTemperatureSensor_t object – the object instance from which the last value of the measurement of the water temperature is to be retrieved to the variable – value. Once the two variables – value and waterTemperatureSensor have been initialized, several assertions are made, the first two are simply verifying that the create() and getLastWaterTemperature() fake functions are called exactly once. The next assertion is an expectation⁶ that the return value of the getLastWaterTemperature() is 0 and this is true as there is no implementation for the function and the default return value for integers is 0. Lastly, the contents of the variable – value are expected to be 0 and this holds true as once again, the variable is assigned the default return type of the function getLastWaterTemperature().

```

/*
 * Message payload Test suite
 */

// Fake function to send a payload message
FAKE_VOID_FUNC(sendMessage, messagePayload);

class Payload_Test : public ::testing::Test
{
protected:
    void SetUp() override
    {
        RESET_FAKE(sendMessage);
        FFF_RESET_HISTORY();
    }

    void TearDown() override
    {
    }
};

```

Figure 17 - Message Payload test class

Figure 17 - Message Payload test class - contains the declaration of the fake function sendMessage() that has as an argument a messagePayload type and it has no return type. In the test class' SetUp(), for each test the sendMessage() fake function is reset so that it can be reused and the fake function call history is reset as well.

⁶ The difference is that upon test failure, the expectation will not halt the rest of the tests within the test suite whereas the assertion will.

```

TEST_F(Payload_Test, Test_create_payload_and_send)
{
    // Arrange
    messagePayload messageToSend;
    messageToSend.len = 4;

    int8_t waterTemperature = 25;
    uint16_t waterLevel = 2000;
    // Act
    messageToSend.bytes[0] = waterLevel >> 8;
    messageToSend.bytes[1] = waterLevel & 0xFF;
    messageToSend.bytes[2] = waterTemperature >> 8;
    messageToSend.bytes[3] = waterTemperature & 0xFF;

    sendMessage(messageToSend);
    // Assert
    ASSERT_EQ(messageToSend.len, 4);
    for (int i = 0; i < messageToSend.len; i++)
    {
        if (i % 2 == 1) // bytes 2 - [1] and 4 - [3], low bytes
        {
            ASSERT_TRUE(messageToSend.bytes[i] == 0xD0); // hex - low byte water level
            ASSERT_TRUE(messageToSend.bytes[i] == 208); // decimal - low byte water level
            ASSERT_TRUE(messageToSend.bytes[3] == 0x19); // hex - low byte water temperature
            ASSERT_TRUE(messageToSend.bytes[3] == 25); // decimal - low byte water temperature
        }

        if (i == 0)
        {
            ASSERT_TRUE(messageToSend.bytes[i] == 0x07); // hex - high byte water level
            ASSERT_TRUE(messageToSend.bytes[0] == 7); // decimal - high byte water level
        }

        if (i == 2)
        {
            ASSERT_TRUE(messageToSend.bytes[i] == 0x00); // hex - high byte water temperature
            ASSERT_TRUE(messageToSend.bytes[i] == 0); // decimal - high byte water temperature
        }
    }
    EXPECT_TRUE(sendMessage_fake.call_count == 1);
}

```

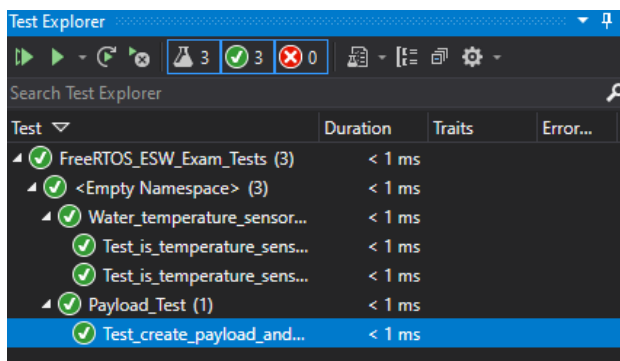
Figure 18 - Message Payload test suite

Figure 18 - Message Payload test suite

shows the mocking of assembling a messagePayload object and sending it.

In the Arrange subsection of the test suite, the variable messageToSend is declared and then its length is initialized to 4 (as it will contain 4 bytes pairs - high/low for the 2 measurements that will be stored in it). Following is the initialization of two variables, waterTemperature and waterLevel with the values of 25 and 1000.

The Act subsection consists of the message payload assembly. The high bytes' bits are shifted right 8 positions and the low bytes' bits are matched with 0b11111111 (0xFF). The Assert subsection's tests verify that all the bytes hold the intended values and by concatenation will result in the initial values. Hence 0x07D0 will result in 2000 (waterLevel) and 0x0019 will result in 25 (waterTemperature). Lastly, the sendMessage function call is evaluated and expected to be 1.



Test	Duration	Traits	Error...
FreeRTOS_ESW_Ext_Tests (3)	< 1 ms		
<Empty Namespace> (3)	< 1 ms		
Water_temperature_sensor...	< 1 ms		
Test_is_temperature_sens...	< 1 ms		
Test_is_temperature_sens...	< 1 ms		
Payload_Test (1)	< 1 ms		
Test_create_payload_and...	< 1 ms		

Figure 19 - Test results

As it can be seen in Figure 19 - Test results, all the tests pass and thus deliver the intended functionality in the end. By this, my meaning is that the test project was used just to briefly understand the necessary collaboration between the assembly, sending and reading of the messagePayload struct, together with usage of ADTs (objects), the interfaces used for the test suites are shown in Figure 20 and Figure 21.

```

typedef struct waterTemperatureSensor* waterTemperatureSensor_t;

/*
 * Function to create the water temperature sensor ADT externally
 * Return: an water temperature sensor object (type: waterTemperatureSensor_t)
 */
waterTemperatureSensor_t waterTemperatureSensor_create();

/*
 * Function to retrieve the last water level from the water temperature sensor
 * Parameters:
 * 1) waterTemperatureSensor_t self - reference to the water temperature sensor ADT from
 *    which the latest water temperature is to be retrieved.
 * Return: the last water temperature (type: int8_t)
 */
int8_t waterTemperatureSensor_getLastWaterTemperature(waterTemperatureSensor_t self);

```

Figure 20 - waterTemperatureSensor_ADT.h (Water temperature sensor module interface)

```

#pragma once

#include <stdint.h>

/*
 * Message payload struct variable
 * Contained variables:
 * 1) len (uint8_t) = length of payload (size)
 * 2) bytes[4] = byte array (type uint8_t)
 */
typedef struct messagePayload {
    uint8_t len;
    uint8_t bytes[4];
}messagePayload;

// Function to send a payload object, parameters: payload - to be sent (type: messagePayload)
void sendMessage(messagePayload payload);

```

Figure 21 - payload.h (Transmitter module interface)

Significant findings and observations

```

#pragma once

// FreeRTOS dependencies
#include <FreeRTOS.h>
#include <task.h>
#include <event_groups.h>
#include <message_buffer.h>
#include <semphr.h>

#include <stdint.h>

```

Figure 22 - Dependencies (configuration.h)

Throughout the development process of the project, I had discovered several interesting facts, by using as a base for the solution the FreeRTOS project template, I spent quite the time trying to create tasks externally to the Main module (main.c). I discovered a way to do this by using a specific dependency sequence in my configuration file (shown in Figure 22) which is used throughout all the task modules (as an included dependency in their interfaces). By using <> (angled brackets – external interface – “made by someone else”), it is possible to create tasks from an external module, in this project solution, in the appController task module and use a dependency pipeline through interfaces, however in my case at least, it was crucial that the first import is <FreeRTOS.h> , otherwise the build process would fail and the executable would not run, this issue was mostly related to the Linker not being able to find the FreeRTOS library files.

Implementation

```
#include "appController.h"

// -----
void main(void)
{
    // Application starting point --> initializes the app controller
    appController_initialize();

    // Let the operating system take over :)
    vTaskStartScheduler();
}
```

Figure 23 - Main module

The application starting point is the Main module (main.c - Figure 23). By having a dependency to the Application controller module interface (appController.h), the Main module can call the `appController_initialize()` function which will initialize⁷ the entire application. The `vTaskStartScheduler()` function is used for the real-time execution and scheduling (control) of tasks by the kernel (complete control of the entire application environment).

```
appController.h  main.c
FreeRTOS_ESW_Exam
1  #pragma once
2  #include "transmitter.h"
3  #include "waterLevelSensor.h"
4  #include "waterTemperatureSensor.h"
5  #include "configuration.h"
6
7  // Function to create the app controller externally
8  void appController_initialize();
9
```

Figure 24 - Application Controller Module Interface

The possibility for the application initialization to occur from an external module is made possible by the use of an interface (appController.h - Figure 24) that exposes to the client – the Main module (main.c) the `appController_initialize()` function. Furthermore, the interface also contains the necessary dependencies to the rest of the application components – modules through the inclusion of their respective interfaces (transmitter.h, waterLevelSensor.h, waterTemperatureSensor.h), the dependency to the configuration file (configuration.h) is used throughout all of the application interfaces as this file contains external dependencies to the FreeRTOS library and configuration macros (task priorities, stack size and bit values for the event group bits).

⁷ Create and initialize the FreeRTOS objects – message buffer, event group, mutex; create and initialize the sensor task ADTs (objects) – water level and temperature sensors; create the transmitter task module; control the measurement event of the water level task; retrieve the measurements (water level and temperature values) from their respective tasks and lastly, assemble and send the message payload to the transmitter task.

```

appControllerTask.c  X
FreeRTOS_ESW_Exam
1  #include "appController.h"
2  /*
3   * Necessary private variable for task synchronization, protect
4   * - app controller task handle
5   * - event group for data ready (measurement finished) synchron
6   * - mutex semaphore for protection of resources (printf stat
7   * - message buffer through which the payload is sent to the
8   * - payload size (maximum possible, as per the documentation)
9   */
10 static TaskHandle_t appController_task_handle = NULL;
11 static EventGroupHandle_t pvEventGroup;
12 static SemaphoreHandle_t mutexSemaphore;
13 static MessageBufferHandle_t payloadMessageBuffer;
14 static const size_t payloadMessageBufferSizeBytes = 10;
15
16 /*
17 * Sensor variables - initialized with NULL (will be created wi
18 */
19 static waterLevelSensor_t waterLevelSensor = NULL;
20 static waterTemperatureSensor_t waterTemperatureSensor = NULL;
21
22 // App controller task function declaration
23 void appController_task(void* pvParameters);
24

```

Figure 25 - Application Controller Module - part 1 (appControllerTask.c)

In Figure 25 , the variables used within the module are shown together with the interface dependency (appController.h). The FreeRTOS objects are the appController_task_handle (the TCB), pvEventGroup (the event group used for synchronization with the water level task module), mutexSemaphore for protection of the print statements (resources) and payloadMessageBuffer (the message buffer to which the message payload will be sent once it is assembled). NULL variables (initially) – waterLevelSensor and waterTemperatureSensor are the variables used to store the instances of the two measurement sensor task modules (these will later hold references to actual instances of the measurement sensor task modules, once the instances are created). The payloadMessageBufferSizeBytes is initialized with the value of 10 (even though there are only 4 bytes that will be sent, as per the documentation of the message buffer object - message_buffer.h an additional 4 bytes is generally added to the size, for ensuring that size will not present an issue, I decided to use 10).

```

void appController_initialize()
{
    pvEventGroup = xEventGroupCreate(); // Create the event group
    mutexSemaphore = xSemaphoreCreateMutex(); // Create the mutex
    payloadMessageBuffer = xMessageBufferCreate(payloadMessageBufferSizeBytes); // Create the message buffer

    // verify if the initializations were successful
    if (pvEventGroup != NULL && mutexSemaphore != NULL && payloadMessageBuffer != NULL)
    {
        waterLevelSensor = waterLevelSensor_create(pvEventGroup, mutexSemaphore); // Create the water level sensor
        waterTemperatureSensor = waterTemperatureSensor_create(mutexSemaphore); // Create the water temperature sensor
        transmitter_create(payloadMessageBuffer, mutexSemaphore); // create transmitter

        // Create the app controller task
        xTaskCreate(appController_task, // Function that implements the task
            (const portCHAR*)"App Controller", // Task name
            TASKS_STACK, // Allocated Stack size (words, not bytes)
            NULL, // Parameters
            APP_CONTROLLER_TASK_PRIORITY, // Priority of the task
            &appController_task_handle); // Task handle
    }
}

```

Figure 26 - Application Controller module - part 2 (appControllerTask.c)

Figure 26 - Application Controller module - part 2 (appControllerTask.c), is the appController_initialize() function body implementation. Here, the previously declared variable are defined – pvEventGroup, mutexSemaphore, payloadMessageBuffer with a size of 10 bytes. The if block verifies that the previously mentioned variables are not null (that they were successfully initialized) and then allows the execution of the application to proceed to the initialization⁸ of the waterLevelSensor, waterTemperatureSensor variables (the measurement sensor task ADT objects) and the call to transmitter_create() that creates the transmitter task module. The last section within the if block is the xTaskCreate() function in which the Application Controller module's task is created (the priority is 2).

```
void appController_task(void* pvParameters)
{
    for (;;)
    {
        // Set the measurement bit (bit 0) in the data ready event group
        xEventGroupSetBits(pvEventGroup, WATER_LEVEL_MEASUREMENT_BIT);
        /* xEventGroupWaitBits function parameters:
         * event group name,
         * the bits that are waited for (bit 1),
         * clear bits before return (pdTRUE - true),
         * wait for the specified bit(s) (pdTRUE - true),
         * wait indefinitely (portMAX_DELAY)
         */
        xEventGroupWaitBits(pvEventGroup, WATER_LEVEL_DATA_READY_BIT, pdTRUE, pdTRUE, portMAX_DELAY);
        /*
         * Retrieve latest measurements of the water level and water temperature from the
         * water level sensor and water temperature sensor
         */
        uint16_t waterLevel = waterLevelSensor_getLastWaterLevel(waterLevelSensor);
        int8_t waterTemperature = waterTemperatureSensor_getLastWaterTemperature(waterTemperatureSensor);
        /*
         * Declare and initialize the payload that will be sent
         * Payload initialization:
         * - .len = 4 (4 bytes to send)
         * - .bytes = {0} (all bytes set to 0)
         */
        messagePayload payloadMessage = { .len = 4, .bytes = {0} };
        // Assemble the payload
        payloadMessage.bytes[0] = waterLevel >> 8; // bit-wise shift right 8 positions
        payloadMessage.bytes[1] = waterLevel & 0xFF; // bit-wise AND with 0b11111111
        payloadMessage.bytes[2] = waterTemperature >> 8; // bit-wise shift right 8 positions
        payloadMessage.bytes[3] = waterTemperature & 0xFF; // bit-wise AND with 0b11111111
        /*
         * Send the payload to the Message buffer
         * Parameters:
         * 1) payloadMessageBuffer - message buffer to which to send the data
         * 2) &message - the message's memory location contents (point to the data content)
         * 3) sizeof(messagePayload) - message size (data length)
         * 4) portMAX_DELAY - wait time (indefinite) for the buffer to block the receiving task (if there are no free buffers)
         */
        xMessageBufferSend(payloadMessageBuffer, &payloadMessage, sizeof(messagePayload), portMAX_DELAY);
        xSemaphoreTake(mutexSemaphore, portMAX_DELAY);
        printf("Payload sent to the transmitter !\n\n");
        xSemaphoreGive(mutexSemaphore);
        vTaskDelay(pdMS_TO_TICKS(1000)); // delay for 1 S
    }
}
```

Figure 27 - Application Controller module - part 3 (appControllerTask.c)

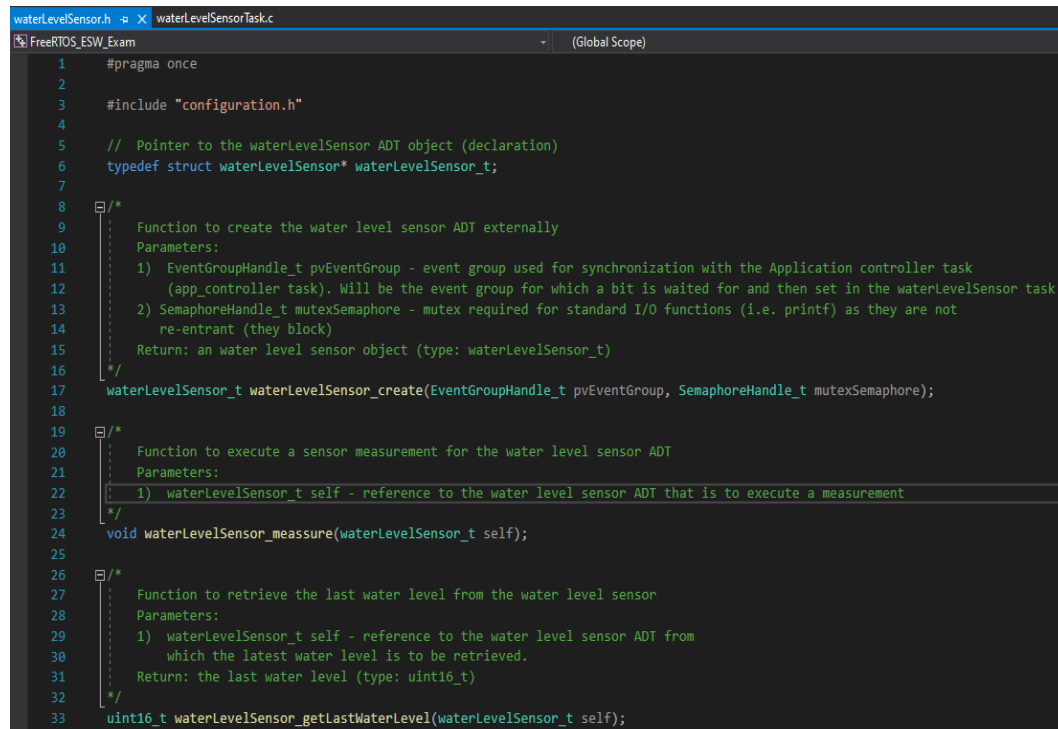
Figure 27, contains the Application control module task body (task definition/critical section). Within this section, the initialization of a measurement that is to be conducted by the Water Level Sensor Task module is done by setting the WATER_LEVEL_MEASUREMENT_BIT (BIT 0) in the event group – pvEventGroup, this bit is waited upon by the Water Level Sensor Task module, once set, the Sensor task will conduct a measurement and set the WATER_LEVEL_DATA_READY_BIT (BIT 1, waited upon

⁸ waterLevelSensor_create(pvEventGroup, mutexSemaphore) – function that returns an instance of waterLevelSensor_t, it takes as arguments an EventGroup_t object – used for synchronization between the Application control module (appControllerTask.c) and the Water Level Sensor module (waterLevelSensorTask.c) and a SemaphoreHandle_t object – used within the Water Level Sensor module for protection of the print statements (resources).

waterTemperatureSensor_create(mutexSemaphore) – function that returns an instance of waterTemperatureSensor_t and takes as an argument a SemaphoreHandle_t object – used within the Water Temperature Sensor module for protection of the print statements (resources).

by the Application Controller Module Task) in the event group to signal that a measurement was completed and the value of the last water level can be retrieved.

Once both water level and water temperature (waterLevel and waterTemperature) values are retrieved from their respective Sensor Module tasks, the message payload (payloadMessage) is assembled, bytes[0] and bytes[2] for the high bytes and bytes[1] and bytes[3] for the low bytes, afterwards, the message payload contents are sent to the message buffer to be later retrieved by the transmitter module task. Lastly, the print statement is protected by the previously created mutex (mutexSemaphore) and the task execution is delayed with 1 second.



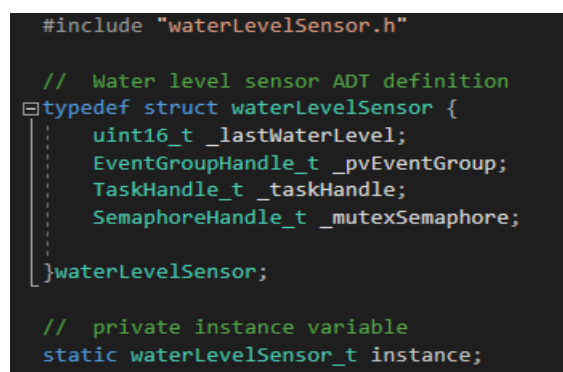
```

1  #pragma once
2
3  #include "configuration.h"
4
5  // Pointer to the waterLevelSensor ADT object (declaration)
6  typedef struct waterLevelSensor* waterLevelSensor_t;
7
8  /**
9   * Function to create the water level sensor ADT externally
10  * Parameters:
11  * 1) EventGroupHandle_t pvEventGroup - event group used for synchronization with the Application controller task
12  *   (app_controller task). Will be the event group for which a bit is waited for and then set in the waterLevelSensor task
13  * 2) SemaphoreHandle_t mutexSemaphore - mutex required for standard I/O functions (i.e. printf) as they are not
14  *   re-entrant (they block)
15  * Return: an water level sensor object (type: waterLevelSensor_t)
16  */
17  waterLevelSensor_t waterLevelSensor_create(EventGroupHandle_t pvEventGroup, SemaphoreHandle_t mutexSemaphore);
18
19  /**
20   * Function to execute a sensor measurement for the water level sensor ADT
21   * Parameters:
22   * 1) waterLevelSensor_t self - reference to the water level sensor ADT that is to execute a measurement
23   */
24  void waterLevelSensor_measure(waterLevelSensor_t self);
25
26  /**
27   * Function to retrieve the last water level from the water level sensor
28   * Parameters:
29   * 1) waterLevelSensor_t self - reference to the water level sensor ADT from
30   *   which the latest water level is to be retrieved.
31   * Return: the last water level (type: uint16_t)
32   */
33  uint16_t waterLevelSensor_getLastWaterLevel(waterLevelSensor_t self);

```

Figure 28 - Water Level Sensor Module interface (waterLevelSensor.h)

In Figure 28, the interface exposes the waterLevelSensor_t type (the water level sensor ADT object declaration) so that it can be used externally by the Application Control module – appControllerTask.c) and the functions create(), measure() and getLastWaterLevel() (waterLevelSensor_ annotated as a prefix so as to be used as a unique identifier/namespace).



```

#include "waterLevelSensor.h"

// Water level sensor ADT definition
typedef struct waterLevelSensor {
    uint16_t _lastWaterLevel;
    EventGroupHandle_t _pvEventGroup;
    TaskHandle_t _taskHandle;
    SemaphoreHandle_t _mutexSemaphore;
} waterLevelSensor_t;

// private instance variable
static waterLevelSensor_t instance;

```

Figure 29 - Water Level Sensor Module part 1

Figure 29 showcases the waterLevelSensor ADT definition, it consists of several variables, _lastWaterLevel to store the value of the measured water level, _pvEventGroup to store the passed event group object from the Application control module to allow synchronization, the task handle _taskHandle and _mutexSemaphore for the protection of print statements (also passed to the ADT from the Application control module through the waterLevelSensor_create() function) and lastly the instance variable that will be returned to the Application Control Module once memory was allocated and the ADT inner variables are set.

```

waterLevelSensor_t waterLevelSensor_create(EventGroupHandle_t pvEventGroup, SemaphoreHandle_t mutexSemaphore)
{
    // Allocate memory for the object
    instance = malloc(sizeof(waterLevelSensor));

    // Verify that memory was available
    if (instance == NULL)
    {
        return NULL;
    }

    // Set the Water level sensor's inner variables
    instance->_lastWaterLevel = 100;
    instance->_pvEventGroup = pvEventGroup;
    instance->_taskHandle = NULL;
    instance->_mutexSemaphore = mutexSemaphore;

    // Create the Water level sensor task
    xTaskCreate(waterLevelSensor_task,
        (const portCHAR*)"Water level task",
        TASKS_STACK,
        NULL,
        WATER_LEVEL_TASK_PRIORITY,
        &instance->_taskHandle);

    return instance;
}

```

Figure 30 - Water Level Sensor Module part 2

Figure 30 - contains the definition of the `waterLevelSensor_create` function, the arguments are the event group and mutex that are being passed by the Application control module once the `waterLevelSensor_create` function is called, then these are set in the ADT object instance. An initial value of 100 is set for the last water level (`_lastWaterLevel`). Within the function body, the Water Level Sensor Task Module task is also created, having a priority of 2 (see `configuration.h`), lastly, the instance – `waterLevelSensor_t` instance is returned (this is used in the Application Control module to initialize the `waterLevelSensor` variable).

```

// Function to execute a measurement on the Water level sensor object
void waterLevelSensor_measure(waterLevelSensor_t self)
{
    self->_lastWaterLevel++;
}

// Water level sensor task function definition
void waterLevelSensor_task(void* pvParameters)
{
    xSemaphoreTake(instance->_mutexSemaphore, portMAX_DELAY);
    printf("Water level sensor task is running\n\n");
    xSemaphoreGive(instance->_mutexSemaphore);
    for (;;)
    {
        /* Wait for the bits to be set;
        parameters:
        - event group variable that is part of the waterLevelSensor,
        - WATER_LEVEL_DATA_READY_BIT - bit that is waited for,
        - clear bits before return (pdTRUE - true),
        - wait for all bits (pdTRUE - true),
        - wait indefinitely (portMAX_DELAY)
        */
        xEventGroupWaitBits(instance->_pvEventGroup, WATER_LEVEL_MEASUREMENT_BIT, pdTRUE, pdTRUE, portMAX_DELAY);

        // execute a measurement on the Water level sensor object
        waterLevelSensor_measure(instance);
        // delay for 100 ms
        vTaskDelay(pdMS_TO_TICKS(100));

        xSemaphoreTake(instance->_mutexSemaphore, portMAX_DELAY);
        printf("Water level --- current (Liters): %d \n\n", instance->_lastWaterLevel);
        xSemaphoreGive(instance->_mutexSemaphore);

        // set the bit in the event group
        xEventGroupSetBits(instance->_pvEventGroup, WATER_LEVEL_DATA_READY_BIT);
    }
}

// Function to return the latest water level measured by the Water level sensor object
uint16_t waterLevelSensor_getLastWaterLevel(waterLevelSensor_t self)
{
    if (self->_lastWaterLevel == 0)
    {
        return 0;
    }
    return self->_lastWaterLevel;
}

```

Figure 31 - Water Level Sensor Module part 3

In Figure 31, the implementation for the measure, task function and getLastWaterLevel are shown.

The waterLevelSensor_measure function is called on the previously defined instance object for symbolizing the execution of measurements from the water level sensor ADT object (the measurement is initially set to 100, then any subsequent measurement will simply hold the previous value incremented by 1). The task body definition showcases the synchronization with the Application Control Module through the event group (_pvEventGroup which was previously initialized with pvEventGroup that was passed as argument upon the creation of the Water Level Sensor ADT object - instance). Each measurement occurs with a delay of 100 milliseconds. And the synchronization with the Application Control module is done once when the WATER_LEVEL_MEASUREMENT_BIT is set in the Application Control module (a measurement is to be done), then the waterLevelSensor_measure() function is called on the instance variable, simulating the execution of a new measurement (the incremented value of _lastWaterLevel within the instance object). The second phase of the synchronization is when a measurement was executed and this is symbolized by setting the WATER_LEVEL_DATA_READY_BIT in the same event group (_pvEventGroup). From this point in the application execution, the Application Control module task can retrieve the _lastWaterLevel variable value of the waterLevelSensor object.

```

#pragma once

#include "configuration.h"

// Pointer to a waterTemperatureSensor ADT object (declaration)
typedef struct waterTemperatureSensor* waterTemperatureSensor_t;

/*
 * Function to create the water temperature sensor ADT externally
 * Parameters:
 * 1) SemaphoreHandle_t mutexSemaphore - mutex required for standard I/O functions (i.e. printf) as they are not
 *    re-entrant (they block)
 * Return: an water temperature sensor object (type: waterTemperatureSensor_t)
 */
waterTemperatureSensor_t waterTemperatureSensor_create(SemaphoreHandle_t mutexSemaphore);

/*
 * Function to retrieve the last water level from the water temperature sensor
 * Parameters:
 * 1) waterTemperatureSensor_t self - reference to the water temperature sensor ADT from
 *    which the latest water temperature is to be retrieved.
 * Return: the last water temperature (type: int8_t)
 */
int8_t waterTemperatureSensor_getLastWaterTemperature(waterTemperatureSensor_t self);

```

Figure 32 - Water Temperature Sensor Module interface (waterTemperatureSensor.h)

In Figure 32 - Water Temperature Sensor Module interface (waterTemperatureSensor.h), the interface exposes the waterTemperatureSensor_t type (the water temperature sensor ADT object declaration) so that it can be used externally by the Application Control module - appControllerTask.c) and the functions create() and getLastWaterTemperature() (waterLevelTemperature_ annotated as a prefix so as to be used as a unique identifier/namespace).

```

#include "waterTemperatureSensor.h"

// Water temperature sensor ADT definition
typedef struct waterTemperatureSensor {
    int8_t _lastWaterTemperature;
    TaskHandle_t _taskHandle;
    SemaphoreHandle_t _mutexSemaphore;
} waterTemperatureSensor;

// private instance variable
static waterTemperatureSensor_t instance;

// Water temperature sensor task function declaration
void waterTemperatureSensor_task(void* pvParameters);

/*
 * Function to create the water temperature sensor ADT object
 * Parameters:
 * 1) SemaphoreHandle_t mutexSemaphore - mutex given by the app controller (used to protect
 *    the Water temperature sensor ADT object
 * Return: the Water temperature sensor ADT object
 */
waterTemperatureSensor_t waterTemperatureSensor_create(SemaphoreHandle_t mutexSemaphore)
{
    // Allocate memory for the object
    instance = malloc(sizeof(waterTemperatureSensor));

    // Verify that memory was available
    if (instance == NULL)
    {
        return NULL;
    }

    // Set the Water temperature sensor's inner variables
    instance->_lastWaterTemperature = 30;
    instance->_taskHandle = NULL;
    instance->_mutexSemaphore = mutexSemaphore;

    // Create the Water temperature sensor task
    xTaskCreate(waterTemperatureSensor_task,
        (const portCHAR*)"Water temperature task",
        TASKS_STACK,
        NULL,
        WATER_TEMPERATURE_PRIORITY,
        &instance->_taskHandle);

    return instance;
}

```

Figure 33 - Water Temperature Sensor Module part 1

The implementation of the Water Temperature Sensor interface (Figure 33 - Water Temperature Sensor Module part 1) begins with the ADT definition, the contained variables in the waterTemperatureSensor struct are _lastWaterTemperature (to hold the value of the last executed measurement of the temperature), _taskHandle (used for the creation of the Water Temperature Module task) and the _mutexSemaphore (to secure/protect the print resources within the module, the mutex is assigned the mutex that will be passed after creation in Application Control module). The instance variable is the one returned to the Application Control Module once memory was allocated and the ADT inner variables are set.

Once the instance variable has its member variables set (initial value of water temperature being 30) and memory was allocated, the Water Temperature Sensor Module task is created (waterTemperatureSensor_task defines the task function body).

```
// Function to execute a measurement on the Water temperature sensor object
static void waterTemperatureSensor_measure(waterTemperatureSensor_t self) {
    self->_lastWaterTemperature++;
}

// Water temperature sensor task function definition
void waterTemperatureSensor_task(void* pvParameters) {

    xSemaphoreTake(instance->_mutexSemaphore, portMAX_DELAY);
    printf("Water temperature sensor task is running\n\n");
    xSemaphoreGive(instance->_mutexSemaphore);

    for (;;)
    {

        // execute a measurement on the Water temperature sensor object
        waterTemperatureSensor_measure(instance);

        // Reset: set _lastWaterTemperature to 1 if _lastWaterTemperature > 50
        if (instance->_lastWaterTemperature > 50)
        {
            xSemaphoreTake(instance->_mutexSemaphore, portMAX_DELAY);
            printf("Water temperature exceeds 50%c --- cooldown initiated\n\n", 248);
            xSemaphoreGive(instance->_mutexSemaphore);
            if (instance->_lastWaterTemperature > 50)
            {
                instance->_lastWaterTemperature = 1;
            }
        }

        xSemaphoreTake(instance->_mutexSemaphore, portMAX_DELAY);
        printf("Water temperature --- current: %d%c\n", instance->_lastWaterTemperature, 248);
        xSemaphoreGive(instance->_mutexSemaphore);
        // delay for 75 ms
        vTaskDelay(pdMS_TO_TICKS(75));
    }

}

// Function to return the latest water temperature measured by the Water temperature sensor object
int8_t waterTemperatureSensor_getLastWaterTemperature(waterTemperatureSensor_t self) {
    if (self->_lastWaterTemperature == 0)
    {
        return 0;
    }
    return self->_lastWaterTemperature;
}
```

Figure 34 - Water Temperature Sensor Module part 2

Figure 34 showcases the waterTemperatureSensor_measure() (static as it is only accessible to the module itself) function that is used to increment the _lastWaterTemperature member variable of the Water Temperature Sensor object instance. Following is the task function body (waterTemperatureSensor_task) in which a measurement is executed every 75 milliseconds and if the value of the _lastWaterTemperature member variable of the Water Temperature Sensor object exceeds the value of 50, it will be assigned the value of 1 ("cooldown initiated"). Lastly, the waterTemperatureSensor_getLastWaterTemperature() is defined to allow function calls on the

Water Temperature Sensor object for retrieving the latest value of the previously executed measurement (`_lastWaterLevel` member variable of the Water Temperature Sensor object instance).

```
1  #pragma once
2
3  #include "configuration.h"
4
5  /* Message payload struct variable
6   * Contained variables:
7   * 1) len (uint8_t) = length of payload (size)
8   * 2) bytes[4] = byte array (type uint8_t)
9   */
10 typedef struct messagePayload {
11     uint8_t len;
12     uint8_t bytes[4];
13 }messagePayload;
14
15 // Function to send a payload object, parameters: payload - to be sent (type: messagePayload)
16 void sendMessage(messagePayload payload);
17
18 /*
19  * Function to create the transmitter externally
20  * Parameters:
21  * 1) MessageBufferHandle_t payloadMessageBuffer - message buffer from which to receive messages (sent payload)
22  * 2) SemaphoreHandle_t mutexSemaphore - mutex required for standard I/O functions (i.e. printf) as they are not
23  *    re-entrant (they block)
24  */
25 void transmitter_create(MessageBufferHandle_t payloadMessageBuffer, SemaphoreHandle_t mutexSemaphore);
```

Figure 35 - Transmitter module interface (transmitterTask.h)

Figure 35 - Transmitter module interface (transmitterTask.h) displays the contents of the Transmitter module interface, the declaration and definition of the `messagePayload` struct variable (used in both the Application control module for assembly of the payload and in the transmitter task for receiving the contents through the message buffer once sent by the Application Control module), the member variables of the `messagePayload` variable are the length of the payload together with an array of bytes (in this case 4 – 2 pairs of high/low bytes for each measurement value – water temperature and level). The two functions declared in the interface are `sendMessage()` that has as an argument a `messagePayload` struct variable, this function will print to the console the contents of the received `messagePayload` once sent through the message buffer. The other function is `transmitter_create()` that is called externally (in the Application Control module) and has two arguments, the `payloadMessageBuffer` (that will be used to establish the synchronization of the Application Control module task and the Transmitter module task and allow the sending-retrieval of message payloads (sending as long as there are any and there is enough space within the message buffer; retrieval as long as there are any payload messages in the message buffer)).

```

#include "transmitter.h"

// Private variables
static MessageBufferHandle_t _payloadMessageBuffer;
static TaskHandle_t transmitter_task_handle = NULL;
static SemaphoreHandle_t _mutexSemaphore;
static messagePayload receivedPayload;

// Transmitter task function declaration
void transmitter_task(void* pvParameters);

/**
 * Function to create the transmitter
 * Parameters:
 * 1) MessageBufferHandle_t payloadMessageBuffer - the message buffer from which data is received
 * 2) SemaphoreHandle_t mutexSemaphore - mutex given by the app controller (used to protect printf sta
 */
void transmitter_create(MessageBufferHandle_t payloadMessageBuffer, SemaphoreHandle_t mutexSemaphore)
{
    // Verify that the buffer and mutex are not null
    if (payloadMessageBuffer != NULL && mutexSemaphore != NULL)
    {
        // Assign the passed message buffer and mutex variables to the local ones
        _payloadMessageBuffer = payloadMessageBuffer;
        _mutexSemaphore = mutexSemaphore;

        // Create the app controller task
        xTaskCreate(transmitter_task, // Function that implements the task
                   (const portCHAR*)"Transmitter task", // Task name
                   TASKS_STACK, // Allocated Stack size (words, not bytes)
                   NULL, // Parameters
                   TRANSMITTER_PRIORITY, // Priority of the task
                   &transmitter_task_handle); // Task handle
    }
}

```

Figure 36 - Transmitter Module part 1

Above, in Figure 36 - Transmitter Module part 1, the Transmitter module implementation is shown. The initial section of the module implementation consists of the module member variables, `_payloadMessageBuffer` is used to store the message buffer object created in the Application Control module (further allowing the synchronization of the two modules' tasks), `transmitter_task_handle` is the TCB⁹ of the transmitter module task, the `_mutexSemaphore` is used to store the mutex object created in Application Control module for protection of the print statements (resources) and lastly, the `receivedPayload` variable is used to store the contents of the sent `messagePayload` once sent through the message buffer.

The function `transmitter_create()` is used externally by the Application Control module to create the transmitter module and its task, the arguments of the function are the `payloadMessageBuffer` and `mutexSemaphore` that are passed by the Application Control Module once the objects are created. The Transmitter module, the task has a priority of 1 (see `configuration.h`), the task body definition is `transmitter_task()`.

⁹ Task control block

```

void transmitter_task(void* pvParameters) {
    xSemaphoreTake(_mutexSemaphore, portMAX_DELAY);
    printf("Transmitter started !\n\n");
    xSemaphoreGive(_mutexSemaphore);

    for (;;)
    {
        // Receive the message (received payload - the contents at the memory address of the variable) in the
        xMessageBufferReceive(_payloadMessageBuffer, &receivedPayload, sizeof(messagePayload), portMAX_DELAY);

        if (receivedPayload.len > 0)
        {
            // Send the received message (receivedPayload, print the contents)
            sendMessage(receivedPayload);
        }
    }
}

/*
Function to send the received payload
Parameters:
1) messagePayload payload - payload to be sent (in our case, just printed to the terminal)
*/
void sendMessage(messagePayload payload)
{
    // Verify that the mutex is not null
    if (_mutexSemaphore != NULL)
    {
        // Take the mutex
        xSemaphoreTake(_mutexSemaphore, portMAX_DELAY);
        printf("Payload was received --- content:\n\n");
        // Print payload contents as long as there are any (byte by byte)
        for (int i = 0; i < receivedPayload.len; i++)
        {
            //printf("Byte[%d]: %X\n", i, receivedPayload.bytes[i]);
            if (i < 2)
            {
                printf("Byte[%d] -- Water level -- : 0x%X\n", i, receivedPayload.bytes[i]);
            }
            else
            {
                printf("Byte[%d] -- Water temperature -- : 0x%X\n", i, receivedPayload.bytes[i]);
            }
        }
        printf("\n\n");
        // Give the mutex (proceed)
        xSemaphoreGive(_mutexSemaphore);
    }
}

```

Figure 37 - Transmitter module part 2

Figure 37 - Transmitter module part 2 shows the Transmitter module task function body definition and the sendMessage() function. The transmitter_task task function body is receiving continuously through the message buffer (_payloadMessageBuffer) payload messages and stores their content in the previously declared receivedPayload variable.

Once a payloadMessage object is sent through the message buffer, its length is evaluated, if greater than 0, then the function sendMessage() is called with the currently receivedPayload as an argument. The sendMessage() function simply displays (prints) the contents of a received payload byte by byte in sequential order with the first two bytes (high-low first pair: bytes[0], bytes[1]) belonging to the water level measurement and last two (high-low second pair: bytes[2], bytes[3]) belonging to the water temperature measurement.