FreeRTOS and emWin hands-on guide

This guide describes the basics of using the FreeRTOS and emWin library. In the following exercises we will learn how to create tasks and communicate between them. We will also see how to build a graphical user interface and display some data on the screen. All examples are written for the MCUXpresso IDE and VisionSOM-RT module.

This is an introduction to the working environment. We will show how to run an example project using the MCUXpresso IDE and VisionSOM-RT module with iMX RT1052 processor.

In the first exercise we will learn how to create and execute tasks in FreeRTOS. We will also see how the tasks can be synchronized.

This example shows the way of interacting between a task and interrupt. We will also learn the meaning of the task priority and its influence on the system.

In this exercise we will use a queue for communication between different tasks.

The fourth exercise will show how to build a simple GUI using the emWin library in order to visualize the measurements data.

In the last example we will show how to add the touch panel support to the emWin library using FreeRTOS software timer.

The example applications will be run on the MiMXRT1052 processor that is part of the VisionSOM-RT module. To simplify the development, the module will be attached to the VisionCB-RT-STD Carrier Board shown on the Illustration 1. The illustration shows also the LCD, JTAG and serial port connectors as well as the I2C pins that will be used in our examples.

During the exercises we will measure the temperature, pressure and acceleration using the KAmodMPL3115A2 and KAmodMMA8541Q modules. They can be connected together to the single I2C bus as it was shown on the Illustration 2. This illustration shows also the connected JTAG programmer and the LCD.

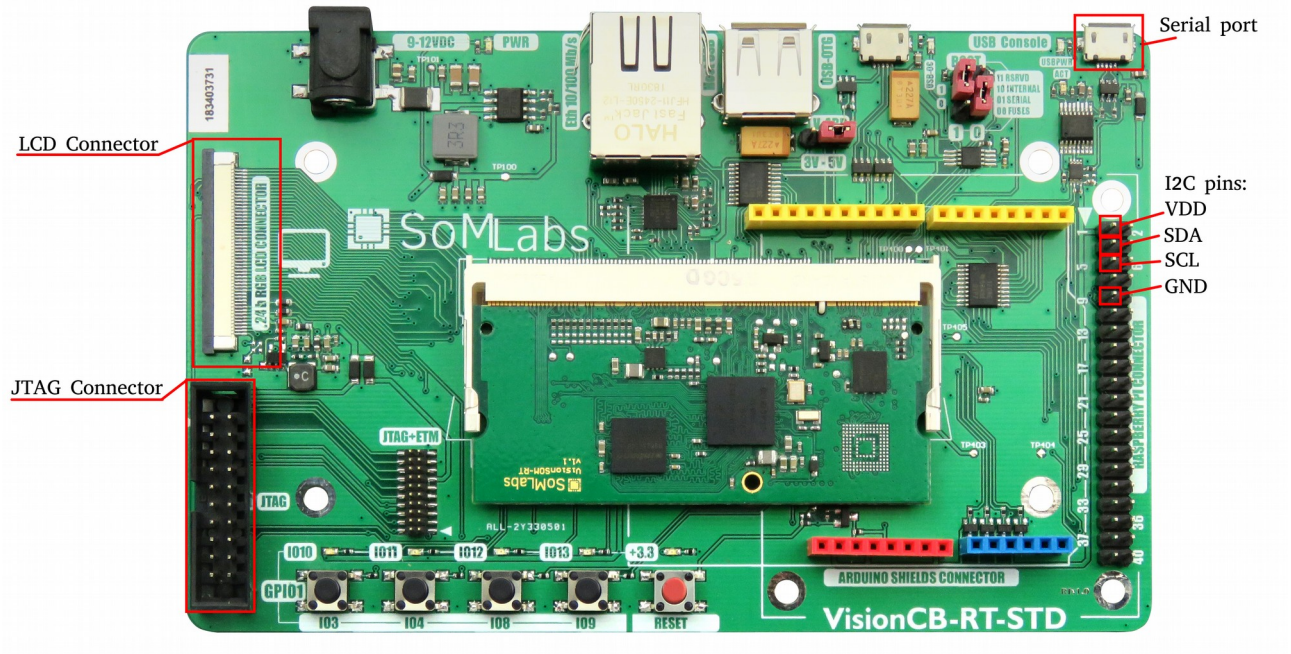

Illustration 1: VisionCB-RT-STD Carrier Board with VisionSOM-RT module

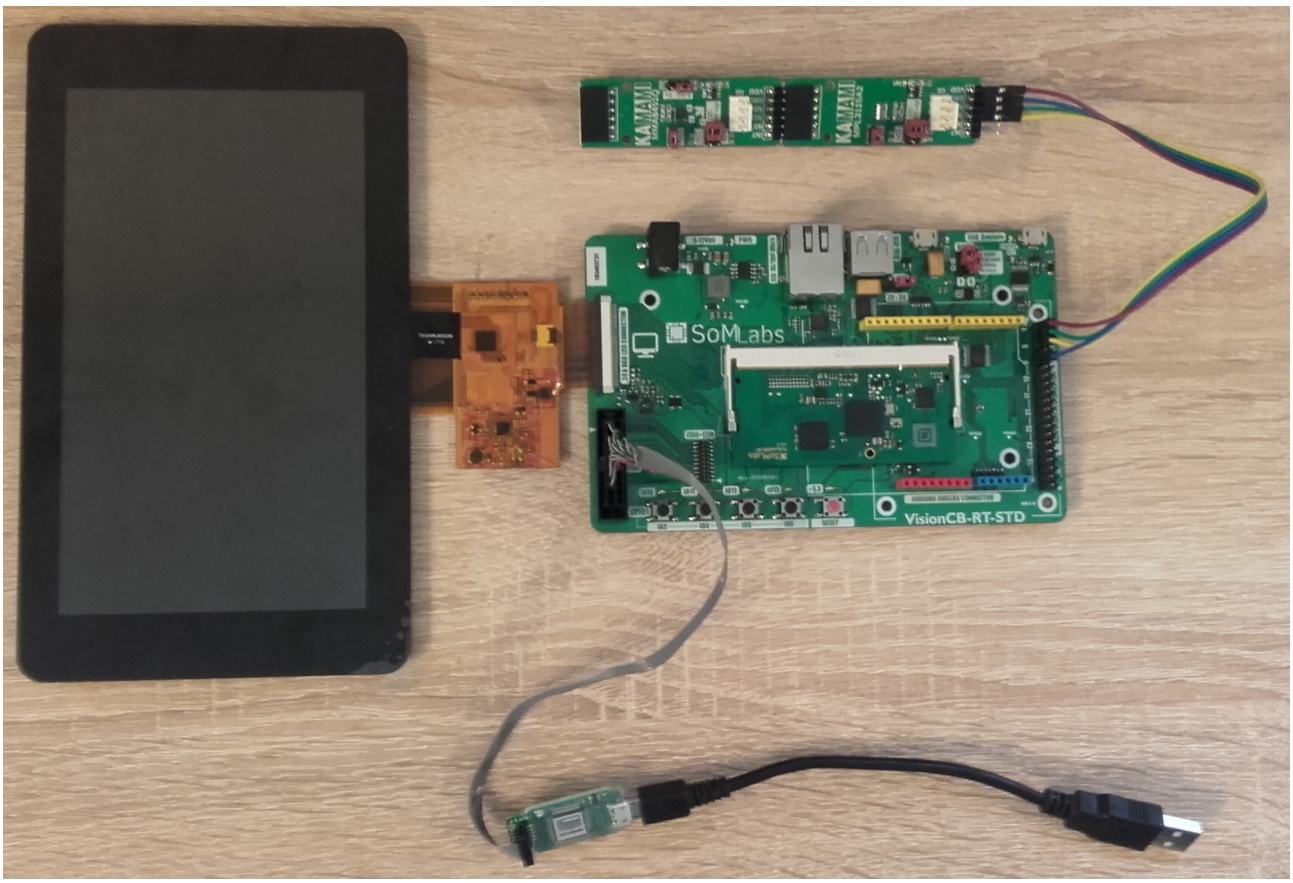

Illustration 2: VisionCB-RT-STD Carrier Board with connected peripherals

Exercise 0

This is the description of the environment that will be used during the workshop. All exercises will be based on the information presented in this chapter so take a closer look at it before going further.

All code examples are prepared for the MCUXpresso IDE v10.3.0. which can be downloaded from the official NXP website using the following link:

https://www.nxp.com/support/developer-resources/software-development-tools/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE

MCUXpresso is based on the Eclipse IDE and supports the following operating systems:

- Microsoft Windows 7/8/10
- Ubuntu Linux 16.04 LTS and later (64-bit host OS only)
- Mac OS X 10.11 and later

There are no other operating system requirements for this workshop.

All source code for the examples is located in the zip archive and structured as it is shown on the Illustration 3. The archive with the project files is located in the *template* directory. The template/VisionSOM_RT_FreeRTOS_emWIN_HandsOn.zip can be imported directly to the MCUXpresso IDE using the *File→Import* option (Illustration 4). This project will be a base for the remaining exercises.

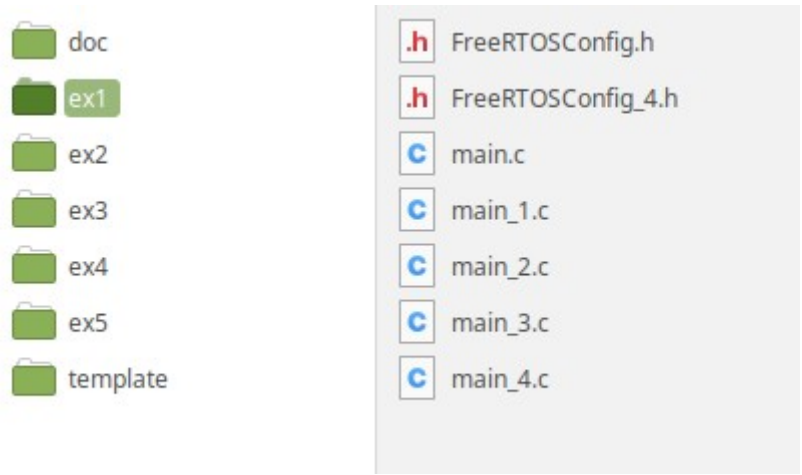

Illustration 3: Hands-on archive structure

Each exercise is located in a single directory containing a base main.c file. This file needs to be copied into the imported project overwriting the existing one. Each main.c file contains a code base that will be explained and expanded during the workshop. Every exercise is also divided into stages serving as checkpoints. The example solution for every stage is located in a separate file with an underscore and a number. We will add the source code to the main.c file, but some exercises will also need some changes in the FreeRTOS configuration located in the FreeRTOSConfig.h file. This file is also added in the original and the changed version for convenience.

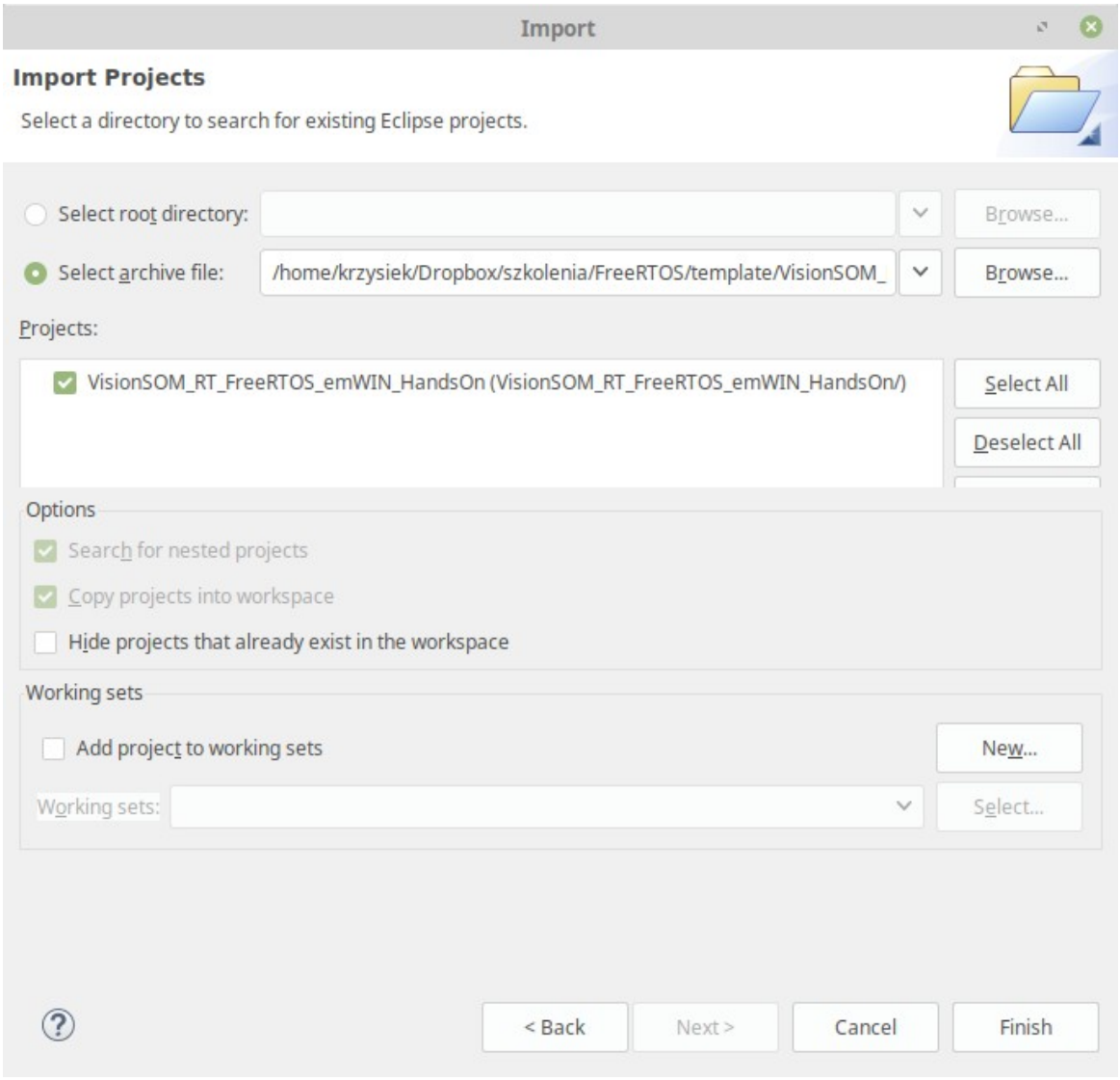

Illustration 4: Importing the project template into the MCUXpresso IDE

For correct project compilation we also need to import the prepared SDK containing the microcontroller libraries. The SDK was build using the official creator available on the NXP website and can be found in the template directory of the hands-on archive. To import the SDK into the MCUXpresso we simply need to drag and drop the SDK_2.5.0_MIMXRT1052xxxxB.zip archive into the *Installed SDKs* window (Illustration 5).

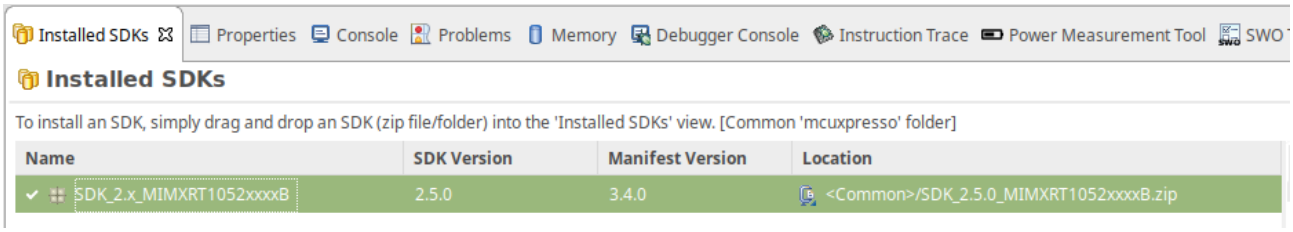

Illustration 5: SDK imported to the MCUXpresso

Each of the example applications will be programmed to the QSPI flash that is located on the module board. In order to do it using the SEGGER J-Link programmer we need to do some changes in the SEGGER JlinkDevices.xml script, that is located in the SEGGER tools installation directory. In the script we need to find the iMXRT105x processor section and set the loader to the QSPI. The example entries look as follows:

```
<!--                 -->
<!-- NXP (iMXRT105x) -->
<!--                 -->
<Device>
    <ChipInfo Vendor="NXP" Name="MCIMXRT1051" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
  </Device>
  <Device>
    <ChipInfo Vendor="NXP" Name="MIMXRT1051xxxxA" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" Aliases="MIMXRT1051xxx5A; MIMXRT1051CVL5A; MIMXRT1051xxx6A;
MIMXRT1051DVL6A" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
  </Device>
  <Device>
    <ChipInfo Vendor="NXP" Name="MIMXRT1051xxxxB" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" Aliases="MIMXRT1051xxx5B; MIMXRT1051CVL5B; MIMXRT1051xxx6B;
MIMXRT1051DVL6B" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
  </Device>
  <Device>
    <ChipInfo Vendor="NXP" Name="MCIMXRT1052" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
  </Device>
  <Device>
    <ChipInfo Vendor="NXP" Name="MIMXRT1052xxxxA" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" Aliases="MIMXRT1052xxx5A; MIMXRT1052CVL5A; MIMXRT1052xxx6A;
MIMXRT1052DVL6A" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
  </Device>
  <Device>
    <ChipInfo Vendor="NXP" Name="MIMXRT1052xxxxB" WorkRAMAddr="0x20000000" WorkRAMSize="0x00080000"
Core="JLINK_CORE_CORTEX_M7" Aliases="MIMXRT1052xxx5B; MIMXRT1052CVL5B; MIMXRT1052xxx6B;
MIMXRT1052DVL6B" />
    <FlashBankInfo Name="QSPI" BaseAddr="0x60000000" MaxSize="0x04000000"
Loader="Devices/NXP/iMXRT105x/NXP_iMXRT105x_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" />
</Device>
```

The details can be found on the official SEGGER wiki page: https://wiki.segger.com/I.MXRT1050

Exercise 1

Let’s start with the first example located in the ex1 directory. First, we need to copy and overwrite the main.c and FreeRTOSConfig.h files. The base source code is shown on the Listing 1. It is responsible for configuring the hardware, creating a single FreeRTOS task and starting the scheduler. The task uses a LPUART driver to write a string to the serial port which can be observed on the PC connected to the *USB Console* connector. For this example the FreeRTOS scheduler is configured for preemption with time slicing enabled, which can be verified by checking the configUSE_PREEMPTION and configUSE_TIME_SLICING values located in the FreeRTOSConfig.h file.

```
TaskHandle_t task1Handle;
void task1(void* param) {
        char* text = "HELLO TASK 1\n";
        while(1) {
            LPUART_WriteBlocking(LPUART1, (uint8_t*)text, strlen(text));
        }
}
int main(void) {
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitBootPeripherals();
    printf("Hello World\n");
    xTaskCreate(task1, "TASK1", 128, NULL, 1, &task1Handle);
    vTaskStartScheduler();
    return 0 ;
}
```

Listing 1: Creating FreeRTOS task

Our first goal is to create another task printing another string to the serial output. It can be achieved by adding another TaskHandle_t variable and calling xTaskCreate function. Of course the implementation of the new task is also needed. The required changes are shown on the Listing 2 an in the main_1.c file.

```
TaskHandle_t task2Handle;
void task2(void* param) {
        char* text = "HELLO TASK 2\n";
        while(1) {
            LPUART_WriteBlocking(LPUART1, (uint8_t*)text, strlen(text));
        }
}
```

```
int main(void) {
    ...
    xTaskCreate(task1, "TASK1", 128, NULL, 1, &task1Handle);
    xTaskCreate(task2, "TASK2", 128, NULL, 1, &task2Handle);
    vTaskStartScheduler();
    ...
}
```

Listing 2: Adding second task

The example serial port output after mentioned changes can be seen on the Illustration 6. It is not correct because two tasks have access to the serial port without any synchronization. We will solve this problem in the next step.

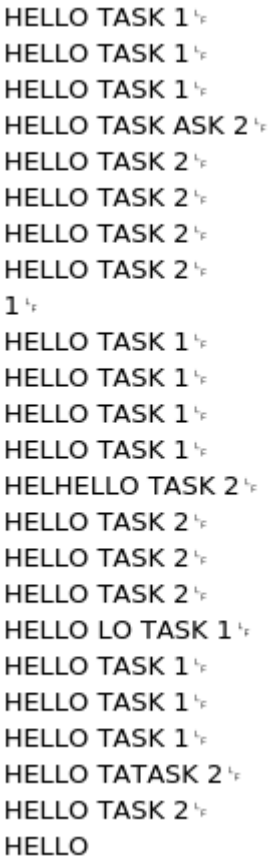

Illustration 6: Example serial port output

Let’s check now how we can add a mutex to fix the output from both tasks. Mutex protects a block of code that may contain a resource that cannot be accessed simultaneously. In our example we need to protect the LPUART_WriteBlocking function call in order to send a complete string before the calling tasks is switched to the other one. The proposed solution can be found in the main_2.c file.

First we need to add the appropriate header file and define a handle that will keep the mutex. It is shown on the Listing 3. The semaphore type and header were not added by mistake – FreeRTOS implements mutex as a special kind of semaphore.

```
#include "semphr.h"

SemaphoreHandle_t mutex;
```

Listing 3: Mutex header and handle declaration

We can create the mutex by calling the xSemaphoreCreateMutex function - in the example it is done in the main function (Listing 4).

```
int main(void) {
    ...
    xTaskCreate(task1, "TASK1", 128, NULL, 1, &task1Handle);
    xTaskCreate(task2, "TASK2", 128, NULL, 1, &task2Handle);
    mutex = xSemaphoreCreateMutex();
    vTaskStartScheduler();
    ...
}
```

Listing 4: Creating the mutex object

Now we can simply add the protection o the task function code as it was shown in the Listing 5. We need to add presented changes in both tasks. The operating system guarantees that the code between the functions xSemaphoreTake and xSemaphoreGive will be fully executed before switching the task as long as both tasks uses the same mutex object.

```
void task1(void* param) {
        char* text = "HELLO TASK 1\n";
        while(1) {
            xSemaphoreTake(mutex, portMAX_DELAY);
            LPUART_WriteBlocking(LPUART1, (uint8_t*)text, strlen(text));
            xSemaphoreGive(mutex);
            taskYIELD();
        }
}
```

Listing 5: LPUART protection using mutex

We also need to understand the taskYIELD call. In the described example, after adding the mutex protection the system scheduler can only switch tasks in the very short time slot between giving the mutex and taking it again. It is only a few processor instructions, so the context switching is very unlikely. Calling taskYIELD informs the scheduler that it can switch the context despite the fact that the currently executing tasks has still some time in the assigned time slot.

After our modifications the output on the serial port should look like the one shown on the Illustration 7.

Illustration 7: Example serial port output after adding mutex protection

So far we heave been using two tasks implemented by two different functions. Now we will see how to use a single function that can be used by two independent tasks. The complete solution is shown in the main_3.c file.

Let’s start from a closer look at the xTaskCreate function. It takes the following arguments:

- task function pointer,
- task name,
- task stack size,
- task parameter,
- task priority,
- task handle.

The given function does not need to be unique , because each task has its own execution context and stack. Moreover, we can pass a pointer to any argument that may be used during task execution. The mentioned approach is implemented in the Listing 6.

```
TaskHandle_t task1Handle;
TaskHandle_t task2Handle;
SemaphoreHandle_t mutex;

char* task1Text = "HELLO TASK 1\n";
char* task2Text = "HELLO TASK 2\n";
```

```
void task(void* param) {
        char* text = (char*)param;

        while(1) {
            xSemaphoreTake(mutex, portMAX_DELAY);
            LPUART_WriteBlocking(LPUART1, (uint8_t*)text, strlen(text));
            xSemaphoreGive(mutex);
            taskYIELD();
        }
}

int main(void) {
    ...
    xTaskCreate(task, "TASK1", 128, task1Text, 1, &task1Handle);
    xTaskCreate(task, "TASK2", 128, task2Text, 1, &task2Handle);
    mutex = xSemaphoreCreateMutex();
    vTaskStartScheduler();
    ...
}
```

Listing 6: Tasks implemented by single function

In this example we have created two tasks that execute the same function. Each task has its own argument, which is a string printed by the LPUART. Besides the argument, the tasks implementation is exactly the same as in the previous version so thanks to the mutex protection the output is like the one already shown on the Illustration 7.

In the last part of this example we will learn about an alternative approach to achieve the correct task synchronization without using any synchronization objects. The required changes are introduced in the main_4.c and FreeRTOSConfig_4.h files.

In the task function we can remove the functions responsible for handling the mutex – xSemaphoreTake and xSemaphoreGive. We will not need them because we will also change the scheduling algorithm. So far we have been using the preemptive scheduler with time slicing. In this version of the code we will remove the time slicing by setting configUSE_TIME_SLICING to 0. In this variant the task will never be preempted by another task with the same or lower priority, so in our example there is no risk that the writing to the serial port will be interrupted by another task. The taskYIELD call is still needed because otherwise the task that will be started as first would never allow the other one to execute. By keeping this call we allow both tasks to execute one after another.

This approach is an example of task synchronization defined by the design of the application. Using the given scheduling algorithm and various task priorities we have full control over the tasks execution.

Exercise 2

In this exercise will learn how to communicate between interrupt and FreeRTOS task using the notification mechanism. We will also see how the tasks priorities may influence the system responsiveness. The source code for this example is shown in the ex2 directory.

We start with the code from the main.c file and FreeRTOS configuration written in the FreeRTOSConfig.h header. It contains a main function which configures the system and creates a single task that turns the LED on and off with a 1000 ms delay. The task function is in the Listing 7.

```
void ledTask(void* param) {
	const TickType_t delayMs = 1000 / portTICK_PERIOD_MS;
	while(1) {
	    GPIO_PinWrite(GPIO1, 8, 1);
	    vTaskDelay(delayMs);
	    GPIO_PinWrite(GPIO1, 8, 0);
	}
}
```

Listing 7: LED handling task

The delay is calculated using a portTICK macro that converts the milliseconds into system ticks used by the vTaskDelay function. The base example code defines also a GPIO interrupt handler which can be triggered by pressing the IO3 button. The handler function clears only the interrupt flags.

Our first goal is to send a notification from the interrupt handler to the task in order to blink the LED after pressing the button. The code is presented in the main_1.c file.

Notifications provide a very simple way of sending single 32-bit value to a specified task. It may be used for synchronization or simple event-based communication. Let’s start with adding the necessary code to the interrupt handler which is shown in the Listing 8. The interrupt handler sends the notification using the xTaskNotifyFromISR function. The value 1 is written overwriting the previous one in case it has not been read. The interrupt handler has also a chance to run a higher priority task than the one that has been executed when the interrupt occurred.

```
void GPIO1_INT3_IRQHandler(void) {
	GPIO_PortClearInterruptFlags(GPIO1, 1U << 3);
	BaseType_t higherPriorityTaskWoken = pdFALSE;
	xTaskNotifyFromISR(ledTaskHandle, 1, eSetValueWithOverwrite, &higherPriorityTaskWoken);
	portYIELD_FROM_ISR(higherPriorityTaskWoken);
}
```

Listing 8: Sending notification from the interrupt handler

The notification is received by the task using the code in the Listing 9. The task execution blocks for a undefined time period on the xTaskNotifyWait function. The first two arguments are the masks of bits that should be cleared before and after the notification value is returned to the task. In the third argument we need to pass the pointer to the 32-bit variable that will contain the notification value and the last one is the notification waiting time. In this example we don't care about the notification value and use it only for the purpose of the synchronization. After running the example we will see the LED (IO10) is turned on for a 1000 ms after pressing the button.

```
while(1) {
    uint32_t notification = 0;
    xTaskNotifyWait(0, UINT32_MAX, &notification, portMAX_DELAY);
    GPIO_PinWrite(GPIO1, 8, 1);
    vTaskDelay(delayMs);
    GPIO_PinWrite(GPIO1, 8, 0);
}
```

Listing 9: Main task loop with the notification reception

In the next step (main_2.c) we will add another task that will use large amount of the processor time. In this implementation it is only a busy loop, however it is a good simulation of some time demanding computations. The task code is presented in the Listing 10.

```
TaskHandle_t busyTaskHandle;
void busyTask(void* param) {
        const TickType_t delayMs = 10 / portTICK_PERIOD_MS;
        char* text = "Calculation finished\n";
        while(1) {
                for(int i = 0; i < 0xFFFFFFF; i++)
                        ;
                LPUART_WriteBlocking(LPUART1, (uint8_t*)text, strlen(text));
                vTaskDelay(delayMs);
        }
}
```

Listing 10: Busy loop task code

The task is created in the main function, but this time let's give it a little higher priority:

```
xTaskCreate(busyTask, "BUSY_TASK", 128, busyTask, 2, &busyTaskHandle);
```

After running this example we can notice that the LED is not turned on immediately after pressing the button, but after finishing the busy loop and writing a information string to the serial port. It is the result of the assigned priorities. The notification is passed immediately from the interrupt to the

ledTask but it needs to wait for the higher priority busyTask to call the vTaskDelay. After this call the scheduler chooses the lower priority task to execute and receive the notification.

To solve this problem we need to increase the priority of the ledTask, like in the main_3.c:

```
xTaskCreate(ledTask, "LED_TASK", 128, ledTask, 3, &ledTaskHandle);
xTaskCreate(busyTask, "BUSY_TASK", 128, busyTask, 2, &busyTaskHandle);
```

Now we can see that pressing the button causes the immediate execution of the ledTask because it is the higher priority task scheduled in the portYIELD_FROM_ISR call.

Last thing that we will check is the influence of the preemption on the tasks behavior. We started this example with preemption and time slicing enabled. Let’s disable the preemption as it is made in the FreeRTOSConfig_4.h file. We can see that the ledTask is again executed just after sending the notification from the interrupt, but the LED is turned off after the busyLoop task enters delay. The low priority task can’t be preempted by the second task, but needs to block by itself.

Exercise 3

Now we can move to the next exercise, where we will learn more about the communication between tasks using queues. The example source code is located in the ex3 directory.

We will start with the code located in the main.c file and the FreeRTOS configuration in the FreeRTOSConfig.h file (preemption and time slicing disabled). As in the previous examples, we will use it as the starting point for the next steps. We have there two tasks: measurementTask and printTask. The first one is responsible for configuring the MPL3115A2 pressure and temperature sensor and reading the values using the I2C bus. The second task generates the text buffer with the measurement results and writes it to the serial port. The only thing that is missing is sending the measured values between these two tasks.

Let's start with adding the required header file and queue handle:

```
#include "queue.h"

QueueHandle_t queueHandle;
```

The queueHandle variable is defined as global because it needs to be visible for both tasks. Now we can create the queue in the main function:

```
queueHandle = xQueueCreate(10, sizeof(uint8_t));
```

The first argument is the maximum number of elements that the queue can hold and the second one is the size in bytes of a single element.

Now, when we have a created queue, we can use it in the tasks. The measurementTask needs to send the data to the queue like in the Listing 11.

```
while(1) {
        vTaskDelay(delayMs);
        FLEXIO_I2C_MasterTransferBlocking(&FlexIO_I2C_1_peripheralConfig, &transfer);

        uint8_t temp = data[3];
        uint32_t press = (data[2] >> 6) | (data[1] << 2) | (data[0] << 10);

        xQueueSend(queueHandle, &temp, 0);
}
```

Listing 11: Enqueueing data

The function xQueueSend pushes the temperature value obtained using the I2C interface to the end of the queue. The data is given by a pointer, however it is copied to the queue according to the element size given during queue creation. The last argument is a time that this function will wait in case the queue is full. If the new element can't be pushed into the queue the xQueueSend function returns errQUEUE_FULL error code. The returned value is not checked in this example in order to keep the source code short and simple.

After enqueueing the temperature value we can receive it in the other task. The code that is able to do this is presented in the Listing 12.

```
while(1) {
        uint8_t temp = 0;
        uint32_t press = 0;

        xQueueReceive(queueHandle, &temp, portMAX_DELAY);

        sprintf(textBuffer, "T = %d, P = %d\n", temp, press);
        LPUART_WriteBlocking(LPUART1, (uint8_t*)textBuffer, strlen(textBuffer));
}
```

Listing 12: Dequeueing data

The data can be pulled from the queue using the function xQueueReceive. The data is copied to given pointer which has to provide enough space for a single element. This function also takes the timeout as a parameter. In this example we wait until a new element is pushed to the queue. We can now run the code which is also located in the main_1.c file and check that the temperature value is written to the serial port.

The queue can hold elements of any type and size. In the previous we were using it for sending simple uint8_t values. Now we will change it to a structure containing both – temperature and pressure readings. Let's start with defining a new type:

```
typedef struct {
        uint8_t temp;
        uint32_t press;
} QueueData_t;
```

It is the type that we want to keep in the queue so we need to change slightly the second argument of the xQueueCreate function:

```
queueHandle = xQueueCreate(10, sizeof(QueueData_t));
```

The queue has still 10 elements, but the single element contains the whole QueueData_t structure so the total amount of memory required by the queue is of course larger than in the previous example. The code responsible for pushing and pulling the data is very similar – we can see that in the Listing 13 and Listing 14.

```
while(1) {
        vTaskDelay(delayMs);
        FLEXIO_I2C_MasterTransferBlocking(&FlexIO_I2C_1_peripheralConfig, &transfer);

        QueueData_t queueData;
        queueData.temp = data[3];
        queueData.press = (data[2] >> 6) | (data[1] << 2) | (data[0] << 10);

        xQueueSend(queueHandle, &queueData, 0);
}
```

Listing 13: Enqueueing the structure

```
while(1) {
        QueueData_t queueData;

        xQueueReceive(queueHandle, &queueData, portMAX_DELAY);

        sprintf(textBuffer, "T = %d, P = %d\n", queueData.temp, queueData.press);
        LPUART_WriteBlocking(LPUART1, (uint8_t*)textBuffer, strlen(textBuffer));
}
```

Listing 14: Dequeueing the structure

Again, we can run the code (main_2.c) and check that this time both – pressure and temperature are visible in the serial port output.

Finally, we will do one more change in our example. Sometimes the data we need to pass from one task to another is so big, that copying each element is unacceptable. In such case we can simply

enqueue a pointer to given data structure, but we always need to remember that this data will be accessed by more than one task, so it needs to be protected by the design of the application or one of the synchronization mechanism. The example code is in the main_3.c file.

First, let’s change the queue creation function call:

```
queueHandle = xQueueCreate(10, sizeof(QueueData_t*));
```

According to the last argument, the size of each element is now equal to the size of a pointer. The changes are also required in the enqueueing and dequeueing functions calls. Last time, in both cases we passed a pointer to the QueueData_t structure. Now we need to pass a pointer to a pointer, like in the Listing 15 and listing 16.

```
QueueData_t queueData;
QueueData_t* queueDataPointer = &queueData;

while(1) {
	vTaskDelay(delayMs);
	FLEXIO_I2C_MasterTransferBlocking(&FlexIO_I2C_1_peripheralConfig, &transfer);

	queueData.temp = data[3];
	queueData.press = (data[2] >> 6) | (data[1] << 2) | (data[0] << 10);

	xQueueSend(queueHandle, &queueDataPointer, 0);
}
```

Listing 15: Enqueueing the structure pointer

```
while(1) {
	QueueData_t* queueData;

	xQueueReceive(queueHandle, &queueData, portMAX_DELAY);

	sprintf(textBuffer, "T = %d, P = %d\n", queueData->temp, queueData->press);
	LPUART_WriteBlocking(LPUART1, (uint8_t*)textBuffer, strlen(textBuffer));
}
```

Listing 16: Dequeueing the structure pointer

Once again we can run the example application and see that the two measured values – temperature and pressure are correctly written to the serial port.

Exercise 4

In this example we will learn how to build a simple user interface for displaying the measurement data using the emWin library. The GUI will be handled by a single FreeRTOS task and will receive the data using a dedicated queue. The source code for this example is in the ex4 directory.

Let’s start with the GUI description. The GUI is built and handed by the guiTask which is presented on the Listing 17.

```
wvoid guiTask(void* param) {

    GUI_Init();
    GUI_Clear();

    const int frameLeft = 20;
    const int frameRight = 5;

    GRAPH_Handle tempGraph = GRAPH_CreateEx(0, LCD_GetYSize() / 2, LCD_GetXSize(),
        LCD_GetYSize() / 2, WM_HBKWIN, WM_CF_SHOW, 0, GUI_ID_GRAPH0);
    GRAPH_SetBorder(tempGraph, frameLeft, 5, frameRight, 5);

    const float tempFactor = 0.125;
    const int tempOffset = -80;

    GRAPH_SCALE_Handle tempScale = GRAPH_SCALE_Create(5, GUI_TA_LEFT, GRAPH_SCALE_CF_VERTICAL, 20);
    GRAPH_SCALE_SetFactor(tempScale, tempFactor);
    GRAPH_SCALE_SetOff(tempScale, tempOffset);
    GRAPH_SCALE_SetTextColor(tempScale, GUI_BLACK);
    GRAPH_AttachScale(tempGraph, tempScale);

    GRAPH_DATA_Handle tempData = GRAPH_DATA_YT_Create(GUI_BLUE,
        LCD_GetXSize() - frameLeft - frameRight, NULL, 0);
    GRAPH_DATA_YT_SetAlign(tempData, GRAPH_ALIGN_LEFT);
    GRAPH_AttachData(tempGraph, tempData);

    char* textBuffer = "     0 mg";

    TEXT_Handle accText = TEXT_CreateEx(500, 110, 150, 50, WM_HBKWIN, WM_CF_SHOW,
        TEXT_CF_RIGHT, GUI_ID_TEXT0, textBuffer)
    TEXT_SetFont(accText, GUI_LARGE_FONT);

    WM_SetDesktopColor(GUI_WHITE);

    GUI_BITMAP bitmap = {
        .XSize = 397,
        .YSize = 96,
        .BytesPerLine = 1191,
        .BitsPerPixel = 24,
        .pData = logoBitmap,
        .pPal = NULL,
        .pMethods = GUI_DRAW_BMP24,
    };

    IMAGE_Handle img = IMAGE_CreateEx(50, 70, 400, 100, WM_HBKWIN, WM_CF_SHOW, 0, GUI_ID_IMAGE0);
    IMAGE_SetBitmap(img,  &bitmap);

    GUI_Exec();

    while(1)
    {
        QueueData_t queueData;
        xQueueReceive(queueHandle, &queueData, portMAX_DELAY);

        GRAPH_DATA_YT_AddValue(tempData, queueData.temp / tempFactor + tempOffset);

        sprintf(textBuffer, "%d mg", queueData.accZ);
        TEXT_SetText(accText, textBuffer);

        GUI_Exec();
    }
}
```

Listing 17: guiTask implementation

The first part of the code is the library initialization. The first call to the emWin library needs to be the GUI_Init function. The only exception is the WM_SetCreateFlags function which sets the default windows creation frags. The next step is to create all GUI components:

- GRAPH for temperature display (with Y axis SCALE),
- TEXT label for acceleration value displaying,
- IMAGE for logo bitmap displaying.

The most complex element is the GRAPH widget that needs the SCALE and DATA objects. These objects need to be separately created and attached to the GRAPH.

Each of the widgets needs to be attached to some other window or widget. It is very convenient approach because we can organize the interface using the relative coordinates. In the example we place all components on the desktop window (WM_HBKWIN) which is always the base component. The coordinates are defined during creation of each object using *_CreateEx functions with the first two arguments – x and y position of the upper left corner of the created object. For the detailed description of the emWin components API please refer to the library guide.

The measurement data is passed to the guiTask using the queue that contains the QueueData_t elements, as it was shown in the previous example. The main loop of the task is responsible for pulling values out of the queue and updating the GUI components: TEXT and GRAPH, by creating a new string with the Z axis acceleration value and adding the temperature to the GRAPH_DATA object. The GRAPH_DATA object takes care about the maximum number of elements (defined in the GRAPH_DATA_YT_Create function call) and removes the oldest point if the internal buffer is full. The GUI_Exec function called at the end of the loop forces the redrawing operation in order to update the image displayed on the screen. The created GUI is shown on the Illustration 8.

Illustration 8: Example user interface

Our task in this step is to add another GRAPH that will display the measured pressure. We will place it next to the existing one, so first let's change the size of our temperature GRAPH. We can do it by changing the third argument of the GRAPH_CreateEx function:

```
GRAPH_Handle tempGraph = GRAPH_CreateEx(0, LCD_GetYSize() / 2, LCD_GetXSize() / 2,
    LCD_GetYSize() / 2, WM_HBKWIN, WM_CF_SHOW, 0, GUI_ID_GRAPH0);
```

It changes the width of the widget, so dividing it by 2 will generate enough space for our new widget. Our temperature GRAPH is now two times shorter, so it has also less space for the visible data. Therefore, let's change the maximum number of points that can be hold by the GRAPH_DATA object by changing the second argument of the GRAPH_DATA_YT_Create function:

```
GRAPH_DATA_Handle tempData = GRAPH_DATA_YT_Create(GUI_BLUE,
    LCD_GetXSize() / 2 - frameLeft - frameRight, NULL, 0);
```

The number of points was originally calculated using the display width and the size of the GRAPH frames, so let's change it to use only half of the LCD size.

Now we can add second GRAPH object to our interface, like in the Listing 18.

```
frameLeft = 30;

GRAPH_Handle pressGraph = GRAPH_CreateEx(LCD_GetXSize() / 2, LCD_GetYSize() / 2, LCD_GetXSize() / 2,
    LCD_GetYSize() / 2, WM_HBKWIN, WM_CF_SHOW, 0, GUI_ID_GRAPH1);
GRAPH_SetBorder(pressGraph, frameLeft, 5, frameRight, 5);

const float pressFactor = 5.0;
const int pressOffset = -100;

GRAPH_SCALE_Handle pressScale = GRAPH_SCALE_Create(5, GUI_TA_LEFT, GRAPH_SCALE_CF_VERTICAL, 20);
GRAPH_SCALE_SetFactor(pressScale, pressFactor);
GRAPH_SCALE_SetOff(pressScale, pressOffset);
GRAPH_SCALE_SetTextColor(pressScale, GUI_BLACK);
GRAPH_AttachScale(pressGraph, pressScale);

GRAPH_DATA_Handle pressData = GRAPH_DATA_YT_Create(GUI_GREEN, LCD_GetXSize() / 2 - frameLeft -
frameRight, NULL, 0);
GRAPH_DATA_YT_SetAlign(pressData, GRAPH_ALIGN_LEFT);
GRAPH_AttachData(pressGraph, pressData);
```

Listing 18: Pressure GRAPH object creation

According to the temperature GRAPH we changed a few things:

- left frame size (frameLeft) was increased to keep the longer Y axis scale labels,
- coordinates of the new GRAPH (LCD_GetXSize() / 2, LCD_GetYSize() / 2),
- SCALE factor and offset defining the values on the Y axis scale.

After creating the widget we also need to update it with the new pressure values in the main loop:

```
GRAPH_DATA_YT_AddValue(pressData, queueData.press / pressFactor + pressOffset);
```

The new version of GUI is shown on the Illustration 9. The source code of the changed version with two GRAPH objects is in main_1.c file.

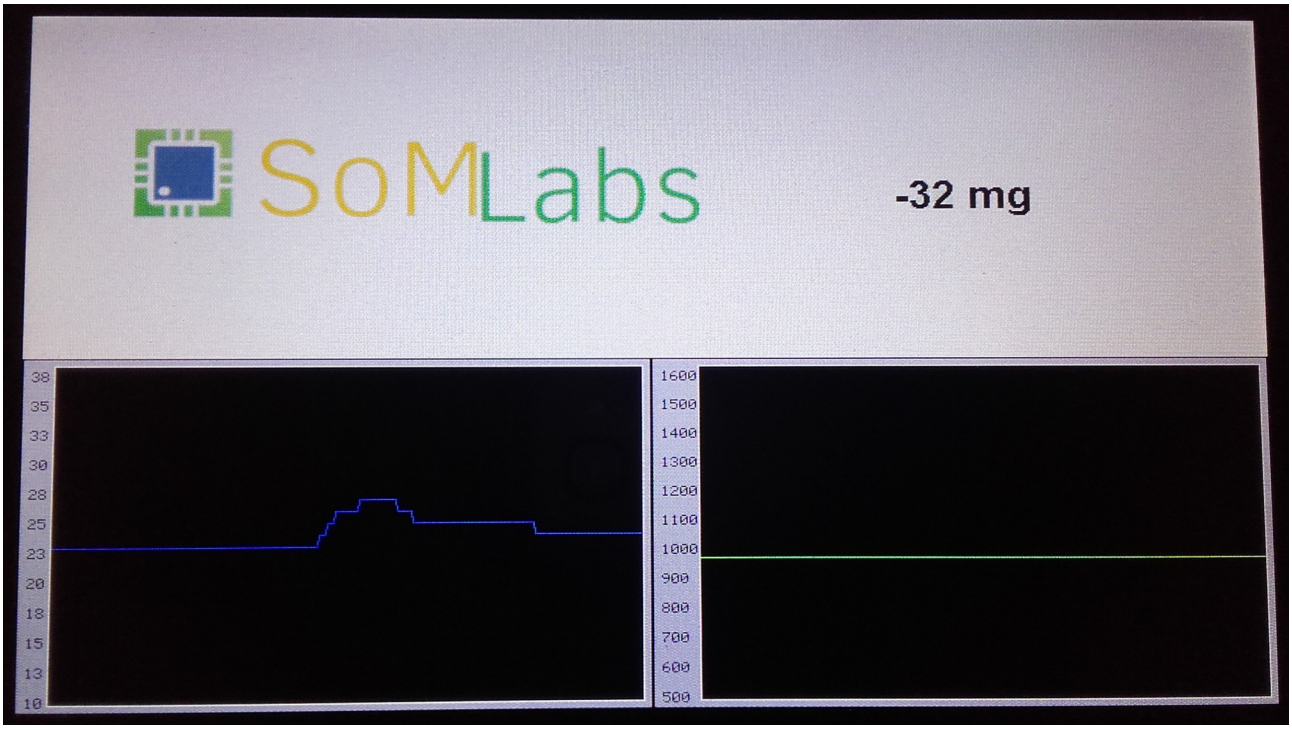

Illustration 9: Interface with two graphs after changes

Exercise 5

In the last example we will implement the support of the touch panel and add some buttons to our interface. We will also need to handle the touch events in the emWin library in order to get notification after user actions. Let’s start with the base implementation in the main.c file in ex5 directory.

In our example we will use a FreeRTOS software timer that will read the current status from the touch screen controller and provide the necessary data to the emWin library. The timers in the FreeRTOS are handled by the timer service task which calls the provided callback functions at specific time. From the implementation point of view, the timer service is like any other task that respects the preemption and time slicing rules. The timer is created in the main function:

```
TimerHandle_t timer = xTimerCreate("TOUCH", 150 / portTICK_PERIOD_MS, pdTRUE, NULL, touchTimer);
xTimerStart(timer, 0);
```

The first function creates the software timer with 150 ms period. The third argument means that the timer will be periodically called by the system. The last two arguments are the timer ID pointer and the callback function. The timer ID is used as an argument for the callback function and can point to any user data. The second function xTimerStart adds the start timer command to the timer service command queue with a specified max blocking time. The timer callback function is shown in the Listing 19.

```
void touchTimer(TimerHandle_t xTimer)
{
	uint8_t data[5] = {0};

	flexio_i2c_master_transfer_t touchTransfer;
	touchTransfer.data = data;
	touchTransfer.dataSize = 5;
	touchTransfer.flags = 0;
	touchTransfer.slaveAddress = 0x38;
	touchTransfer.subaddressSize = 0x01;
	touchTransfer.subaddress = 0x02;
	touchTransfer.direction = kFLEXIO_I2C_Read;
	FLEXIO_I2C_MasterTransferBlocking(&FlexIO_I2C_1_peripheralConfig, &touchTransfer);

	uint8_t points = data[0];
	uint32_t x = ((data[1] & 0x0F) << 8) | (data[2]);
	uint32_t y = ((data[3] & 0x0F) << 8) | (data[4]);

	static bool activeTouch = false;

	GUI_PID_STATE state;
	state.Layer = 0;
	state.Pressed = 0;
	state.x = x;
	state.y = y;

	if (points == 1) {
		activeTouch = true;
		state.Pressed = 1;
		GUI_PID_StoreState(&state);
	} else if (activeTouch) {
		activeTouch = false;
		state.Pressed = 0;
		GUI_PID_StoreState(&state);
	}
}
```

Listing 19: Timer callback function

First, the callback function reads the state of the touch controller FT5426. The received data contains number of detected points, and their coordinates. In the example we use only single point

events for the PID (Pointer Input Device) interface. After reading the data from the controller a GUI_PID_STATE structure is created. It contains the touch coordinates and pressed flag. The pressed flag needs to be set at the beginning of the touch event, and reset at the end of it. During the movement the touch point coordinates are periodically provided to the library. There is also a layer field which represents the layer of the input event. It is usually set to 0. After setting all fields of the structure, it is passed to the library using the GUI_PID_StoreState function.

There is also a small change in the guiTask code according to the previous example. A new WINDOW widget was created in order to process some of the events. The event callback can be set using the function:

```
WM_SetCallback(window, eventCallback);
```

The callback function will receive all events that are intended for the given window. The callback function is empty, but we will implement it in the next step.

Let’s add now some buttons that will allow us to change the axis of the displayed acceleration. We can do it in the guiTask together with other components and attach them to the specially created WINDOW widget:

```
buttonX = BUTTON_CreateEx(670, 40, 100, 50, window, WM_CF_SHOW, 0, GUI_ID_BUTTON0);
BUTTON_SetText(buttonX, "X");
buttonY = BUTTON_CreateEx(670, 100, 100, 50, window, WM_CF_SHOW, 0, GUI_ID_BUTTON1);
BUTTON_SetText(buttonY, "Y");
buttonZ = BUTTON_CreateEx(670, 160, 100, 50, window, WM_CF_SHOW, 0, GUI_ID_BUTTON2);
BUTTON_SetText(buttonZ, "Z");
```

The buttons handles are declares as global variables because we need to access them in the event callback function that is shown on the Listing 20.

```
void eventCallback(WM_MESSAGE * pMsg)
{
	if(pMsg->MsgId == WM_NOTIFY_PARENT) {
		if (pMsg->hWinSrc == buttonX) {
			int code = pMsg->Data.v;
			if (code == WM_NOTIFICATION_RELEASED) {
				activeAxis = ACTIVE_AXIS_X;
			}
		}
		else if (pMsg->hWinSrc == buttonY) {
			int code = pMsg->Data.v;
			if (code == WM_NOTIFICATION_RELEASED) {
				activeAxis = ACTIVE_AXIS_Y;
			}
		}
		else if (pMsg->hWinSrc == buttonZ) {
			int code = pMsg->Data.v;
			if (code == WM_NOTIFICATION_RELEASED) {
				activeAxis = ACTIVE_AXIS_Z;
			}
		}
		return;
	}

	WINDOW_Callback(pMsg);
}
```

Listing 20: Timer callback function

Because the callback function receives every kind of event dedicated to our window, we need to filter the ones that we are interested in. The type of event coming from clicked buttons is WM_NOTIFY_PARENT. Knowing that this is the one that was passed to our callback we can check which button is the source of this event by checking the pMsg→hWinSrc field and

comparing it to each of the buttons handles. The next information we can obtain is whether the button was clicked or released by checking the event message value. When we are sure that the event is the one we are looking for we set the helper variable that holds the active axis that is displayed on the screen. We could of course create another queue that would hold the button events or simply notify the appropriate task. At the end of the function the WINDOW_Callback function is called. It is very important because we need to call the default event handler for each event that was not processed by our code.

The last step is to change the implementation of the guiTask main loop. So far it displayed the Z axis acceleration, but now we are checking the activeAxis helper value and read one of the acceleration fields from dequeued structure, as it is shown in the Listing 21.

```
while(1)
{
    QueueData_t queueData;
    xQueueReceive(queueHandle, &queueData, portMAX_DELAY);

    GRAPH_DATA_YT_AddValue(tempData, queueData.temp / tempFactor + tempOffset);
    GRAPH_DATA_YT_AddValue(pressData, queueData.press / pressFactor + pressOffset);

    int16_t acceleration = 0;
    if (activeAxis == ACTIVE_AXIS_X) {
        acceleration = queueData.accX;
    } else if (activeAxis == ACTIVE_AXIS_Y) {
        acceleration = queueData.accY;
    } else if (activeAxis == ACTIVE_AXIS_Z) {
        acceleration = queueData.accZ;
    }

    sprintf(textBuffer, "%d mg", acceleration);
    TEXT_SetText(accText, textBuffer);
    GUI_Exec();

}
```

Listing 21: Main guiTask loop

The full source code of the final version is in the main_1.c file and the results can be seen on the Illustration 10.

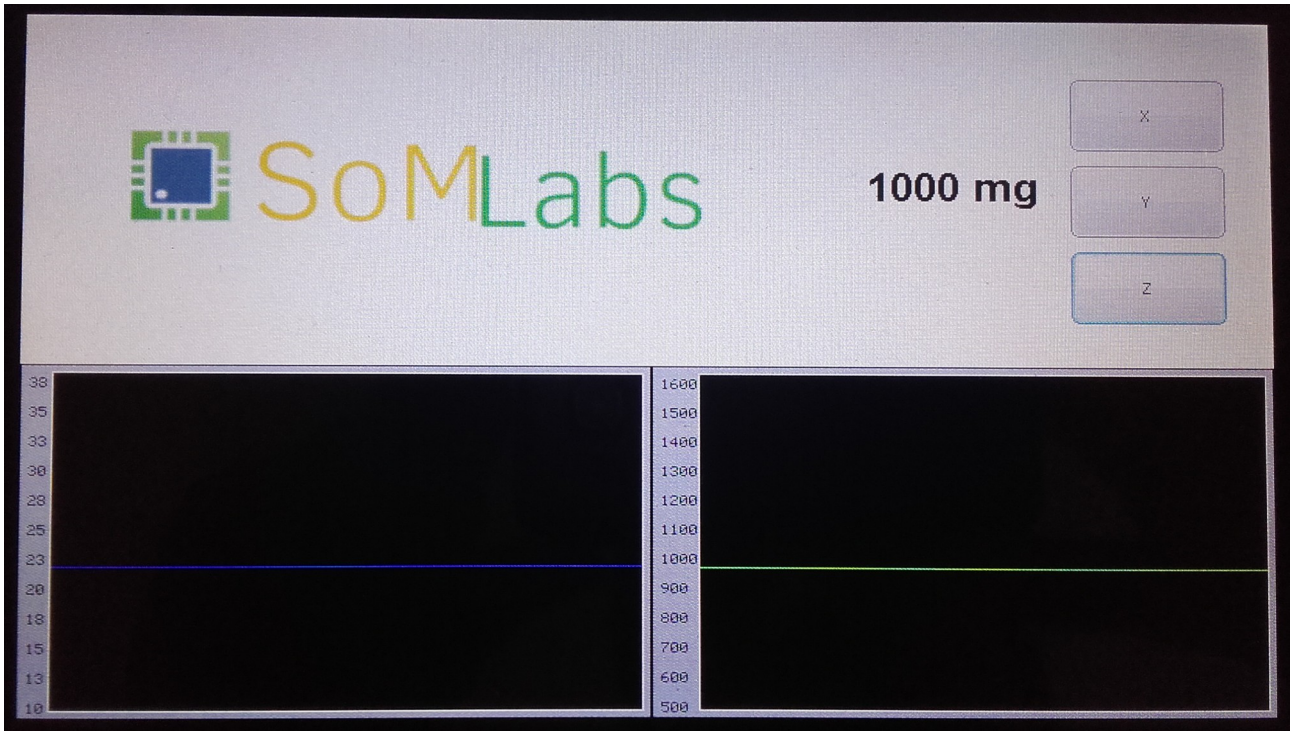

Illustration 10: User interface with buttons