VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Boli Gao

DESIGN OF A RTOS TEACHING ENVIRONMENT

Technology and Communication
2020

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Program of Information Technology

ABSTRACT

Author	Boli Gao
Title	Design of RTOS Teaching Environment
Year	2020
Language	English
Pages	53
Name of Supervisor	Jukka Matila

The thesis is aiming to develop a new teaching environment for a Real-Time Operating System (RTOS) course and design some practical exercises based on the environment. The current teaching method is running μC/OS-II exercises on a personal computer (PC), which is not the case in industries. The new environment first upgrades the kernel used for teaching from μC/OS-II to μC/OS-III and adopts a development board (32F429IDISCOVERY) as a running platform for the μC/OS-III instead of a PC.

After the selections of a RTOS kernel and a development board, a programming environment is also necessary to be built and the preferred solution is using a VScode extension, PlatformIO and a microcontroller configuration tool, STM32CubeMX. Meanwhile, the design of all exercises is primarily focusing on the basic and most important aspects of RTOS, such as Preemptive Scheduling, Task Synchronization and Resource Management and as the exercises need to be easily programmed and tested on the environment, only three peripherals (LED, UART and LCD) are included. Finally, the teaching environment and exercises meet the requirements of the RTOS lecturer but there are still some drawbacks, for example, the exercises can be more refined to cover more concepts of RTOS.

Keywords Real-Time Operating System, μC/OS-III, STM32F429ZI

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF SNIPPETS

LIST OF APPENDICES

1 INTRODUCTION

1.1 Background

Real-Time Operating System (RTOS) now is applied in many areas with the unprecedented development of embedded system from big industries of aerospace, industrial control system, automotive electronics to newly emerging markets such as smart home. It is quite common now that people are using it without noticing the existence of it. As a student whose major is Embedded System, it is important to study RTOS and learn how to apply it in an embedded system project because many employers now require the skills of RTOS and the study of it can contribute to a successful career. However, the teaching environment of RTOS course in Vaasa University of Applied Science (VAMK) is relatively outdated. The RTOS kernel referenced during teaching is μC/OS-II which was first introduced at 1998. The exercises of μC/OS-II are also based on PC, which is not the case in industry.

1.2 Purpose

The first objective of the thesis is to introduce a new version of μC/OS, μC/OS-III, to the lecturer of RTOS course and illustrate the advantages of it over the old version and the benefits from using it. Secondly, it will introduce an embedded teaching environment based on the development board 32F429IDISCOVERY and reveal its high price-performance ratio that is desirable for not only an RTOS course, but other courses of Embedded System as well. The last and most significant objective is to design some μC/OS-III exercises running on 32F429IDISCOVERY. The mandatory exercises should highlight the most basic and important RTOS concepts students need to learn and should also be easy for students to implement while the optional exercises interpret some advanced features that are a little difficult for students to understand.

1.3 Overall Structure

This thesis consists of seven chapters. The first chapter is the introduction which includes some background information, the reasons for writing the thesis and the overall structure of the thesis. The second chapter mainly introduces all components and development tools involved in the thesis and the reasons for choosing one component instead of other available alternatives. Chapter 3 is an instruction to the PlatformIO configuration while Chapter 4 contains an instruction of how to make a μC/OS-III library and an analysis of some porting files. A clean and robust RTOS teaching and learning environment will be introduced at the end of this chapter. Chapter 5 contains the description of all exercises I designed and, the concepts illustrated with these exercises, the philosophy behind the sequence of the exercises and their testing results. In Chapter 6, are conclusions, reflections on what was done in the thesis to see if there are any possible improvement that can be done in the future.

2 COMPONENTS AND DEVELOPMENT TOOLS

2.1 Components

2.1.1 μC/OS-III

μC/OS-III is a highly portable and scalable preemptive real-time kernel. It is the third version of the μC/OS that meets all needs expected from a modern real-time kernel, for instance, task synchronization, task communication, shared resources protection and so on. Meanwhile, it also offers some special features that other RTOSs do not have, for example, direct inter-tasks conversation, run-time configuration and more. Its reputation is not only from its amazing incredible performance, but also from its continuous unified programming style. [1]

What is more exciting is that on 1-March 2020, the license of μC/OS-III changed to "Apache – 2.0", which means it is now an open-source RTOS. The reason why this RTOS is selected is not just because it is the new version of μC/OS-II, which is what currently used in the RTOS course but because it is better than other RTOSs, such as FreeRTOS. FreeRTOS has been an open-source RTOS from the very beginning and that is why it occupies the biggest share of RTOS market. They, similarly, to many other RTOSs, offer quite similar services but with different implementations. For example, the context switching in both RTOSs depends on Pendable Service (PendSV) interrupt and in FreeRTOS, the interrupt service routine will calculate which task has the highest priority and switches to that task but in μC/OS-III, the calculation happens before the interrupt and therefore it spends less time on the interrupt handling. In general, the performance of μC/OS-III on the aspect of "real-time" is stronger than FreeRTOS though FreeRTOS offers some extra functions. Meanwhile, as a mature commercial RTOS, μC/OS-III has more security certifications than FreeRTOS and the security issues are what really need to be considered when people choose a RTOS.

As the exercises involve many new features of μC/OS-III, it is necessary to compare it with the second version. The third version is more powerful than the second version, as Table 1 below shows.

Table 1. The differences between μC/OS-II and μC/OS-III

Feature	μC/OS-II	μC/OS-III
Year introduced	1998	2009
Maximum number of tasks	255	Unlimited
Number of tasks at each priority level	1	unlimited
Round robin scheduling	No	Yes
Mutual exclusion semaphores	Yes	Yes (nestable)
Message mailboxes	Yes	N (not needed)
Signal a task without requiring a semaphore	No	Yes
Option to post/signal without scheduling	No	Yes
Send messages to a task without requiring a message queue	No	yes
Task suspend/resume	Yes	Yes (nestable)
Run-time configurable	No	Yes
Built-in performance measurements	Limited	Extensive
Built-in trace points	No	Yes
Time stamps on posts	No	Yes

Optimizable scheduler in assembly language	No	yes

2.1.2 ARM Cortex-M4 Processor

Before the introduction of the development board, it is necessary to introduce the processor on the board and that is popular ARM Cortex-M4 processor.

The Cortex-M4 processor features low cost, good performance and energy saving. It is designed to target digital signal control industry, so it has an exceptional signal process ability and the combination of these features meets the requirements of many industries, which is why it has a huge developer community. [2]

In fact, ARM has done great effort on the design of Cortex-M series to make it better support porting and running of a RTOS. At least 30 RTOSs now support Cortex-M series and the number is still increasing. The following features make it a suitable platform for μC/OS-III:

1. MSP & PSP:
   The Main Stack Pointer (MSP) is exclusively used for kernel or system exceptions while The Process Stack Pointer (PSP) is used for tasks.
2. NVIC:
   The Nested Vectored Interrupt Controller (NVIC) benefits the running of a RTOS by reducing interrupt latency.
3. SysTick:
   It is a 24-bit built-in system timer that counts down from a reload value saved in SYST_RVR register to zero. It makes time management in a RTOS easy and straightforward because people no longer need to initialize a normal timer to keep the track of system time.
4. PendSV Exceptions:

The enabled Pended Service (PendSV) indicates to the system that an exception needs to be taken care of after all other exceptions or interrupts completed. In an RTOS environment, it is usually used to perform context switching with the lowest priority level.

5. LDREX and STREX

   The Load Register Exclusive (LDREX) and Store Register Exclusive (STREX) instructions perform reading and storing data in memory in some special ways that can be used to implement semaphore or mutual exclusive semaphore in a RTOS.

6. Memory Protection Unit (MPU)

   The Memory Protection Unit is a special unit in the processor that allows a RTOS kernel under privileged mode to grant access permissions of memory. It effectively increases robustness of a RTOS

7. Floating Point Unit (FPU is only available in Cortex-M4)

   It is a big advantage of Cortex-M4 over M3. It not only increases the calculation speed of floating point numbers, but also curtails the consumption of time used for context switching in μC/OS-III.

2.1.3 32F429IDISCOVERY

The STM32F429ZI Discovery Kit takes many advantages of the performance of Cortex-M4 processor and the capability of STM32F429 microcontroller. It is extremely easy for customers to set up its peripherals and with the LCD module; customers can easily build rich graphical applications. [3]

Meanwhile, another possible alternative is the Nucleo-F446RE development board, which is cheaper but has less peripherals than 32F429IDISCOVERY. A favorable feature of this board is its Arduino-shields compatibility while 32F429IDISCOVERY is not. Table 2 below shows the comparison.

Table 2. The comparison between Nucleo-F446re and 32F429IDISCOVERY

	Nucleo-F446re	**32F429IDISCOVERY**
Flash Memory	512KB	2MB
TFT LCD	No	Yes
3-Axis Digital Output Gyroscope	No	Yes
USB OTG with Micro-AB Connector	No	Yes
User LED	1	2
Arduino-Shield Compatible	Yes	No

2.2 Development Tools

2.2.1 PlatformIO for VScode

PlatformIO is a platform-independent ecosystem for embedded development. Its core is written in Python and since Python can be easily run on any operating systems, the project of PlatformIO then is highly portable. Furthermore, PlatformIO can even be running on a server without a graphical desktop because its core is literally a terminal program and people can integrate it with any IDE or editor they like. [4]

Now the recommended solution is the integration with VScode and it has the following features:

1. PIO Unified Debugger: it supports almost all popular debugging probes and more than 600 development boards.
2. Built-in Terminal: it supports the tools of command line interface (CLI) of PlatformIO Core.
3. Library Manager: it makes the inclusion of the official or third-party libraries extremely simple for users.
4. Multiple frameworks: they decide the libraries to be included at the time of the project creation and provide simple examples for testing.
5. PIO Unit Testing: it allows users to easily test different parts of the project individually.

2.2.2 STM32CubeMX

STM32CubeMX is a powerful graphical tool that allows people to easily configure all peripherals or system clock of STM32 microcontrollers and generates the corresponding source code from the configuration and a selected development environment.

Almost all exercises in the thesis only use several lines of UART source code generated by STM32CubeMX but the exercise of displaying text file on the LCD requires a precise configuration of the system and a peripheral clock and it was this STM32CubeMX heavily relied on. However, with the aim of saving more of the lecturer's time for the teaching of RTOS kernel, all source code of initializations will be directly provided to students and then they will not waste any time on the low layer of the learning environment.

3 PLATFORMIO CONFIGURATION

A completed PlatformIO configuration requires the following steps:

1. Install VScode extension “PlatformIO IDE” and it will automatically install all dependent tools (C/C++ extension, GCC Arm Toolchain, OpenOCD, and STM32Cube official libraries). All files are stored in the following folder: C:\Users\USERNAME\.platformio
2. After the installation, there will be an “alien” icon on the left side bar just below the extension icon and by clicking it, quick accesses will be obtained to all services provided by PlatformIO. Now, a new project is made by clicking “open” under “PIO Home” and after that, naming the project and choosing the board (32F429IDISCOVERY in this case). The third selection “Framework” decides what library is going to be included in the project and in this case, it should be “STM32Cube”. The last step is to choose a proper location and click “Finish”.
3. The newly created project will have the following structure:

Figure 1. The typical structure of PlatformIO project.

The “.pio” folder contains all objects and binary files after building the project. The “.vscode” folder includes three json files that are responsible for controlling the build and debug processes of the project and it is not recom-

mended to manually modify these files because they are automatically generated by PlatformIO. As usual the “include” folder is the place to store the project’s header files and the “src” folder stores the source code. The “lib” folder is where the project library is stored but not official libraries because as mentioned, they are all stored in the PlatformIO folder. When Git is used, a version control system, the “.gitignore” file is a useful text file that indicates Git which files or folders in the project to ignore. The “.travis.yml” file is a YAML format text file holding the configuration of Travis CI, which is a hosted continuous integration service used to build and test software projects hosted at GitHub. The “platformio.ini” file is the project configuration file that allows user to manually adject the build environment of the project. The exercises in the thesis only require modification of this file and students will also be encouraged to explore the use of this file.

4. A new project usually will automatically include a BSP (Board Support Package) according to your selection during the creation the project. However, in this case, due to an entry missing in the file “variants_remap”, which is supposed to map the board name created by PlatformIO to its BSP name created by ST, a project of the board 32F429IDISCOVERY will not include its BSP. To fix it, the modification of the file “variants_remap” is necessary. The file is found in the following path:
   C:\Users\USERNAME\.platformio\packages\framework-stm32cube \platformio\variants_remap
   The following entry needs to be added to the file:
   "disco_f429zi": "STM32F429I-Discovery"
5. The LCD library is also in the BSP of 32F429IDISCOVERY but it cannot be used normally until a modification is made in the file “platformio.ini”. The library is dependent on the ili9341 LCD driver, so it is necessary to add the following library dependency in “platformio.ini”:
   “lib_deps = BSP-ili9341”
   Otherwise the ili9341 driver will not be compiled and an undefined reference error will occur when BSP_LCD functions are called.

6. To be able to use ST-link as a UART link, a virtual communication port driver needs to be installed from the site:
   https://www.st.com/en/development-tools/stsw-stm32102.html
7. Reload VScode.

4 μC/OS-III PORTING

4.1 Library Construction

A Successful μC/OS-III porting requires the following steps:

1. Download three repositories from the sites:
   https://github.com/SiliconLabs/uC-OS3
   https://github.com/SiliconLabs/uC-LIB
   https://github.com/SiliconLabs/uC-CPU
2. Make three new folders: “uCOS_Src”, “uCOS_CPU” and “uCOS_Ports”.
3. Copy the files from the folders below to the folder “uCOS_Src”:
   a. /uC-OS3-master/Source:
      All files except for _dbg_uCOS-III.c
   b. /uC-OS3-master/Cfg/Template:
      os_cfg.h
      os_cfg_app.h
   c. In the folder /uC-LIB-master:
      lib_def.h
4. Copy the files from the folders below to the folder “uCOS_Ports”:
   a. /uC-OS-master/Ports:
      os_cpu_c.c
   b. /uC-OS3-master/Ports/ARM-Cortex-M/ARMv7-M/GNU:
      os_cpu.h
      os_cpu_a.s
5. Copy the files from the folders below to the folder “uCOS_CPU”:
   a. /uC-CPU-master:
      cpu_core.c
      cpu_core.h
      cpu_def.h
   b. /uC-CPU-master/ARM-Cortex-M/ARMv7-M:

cpu_c.c

c. /uC-CPU-master/ARM-Cortext-M/ARMv7-M/GNU:

cpu.h

cpu_a.s

d. /uC-CPU-master/Cfg/Template:

cpu_cfg.h

6. Create a new project in PlatformIO and copy these folders to “lib”.
7. If the project is compiled now, an “undefined error” will occur. Open the file “cpu_cfg.h” in the folder “uCOS_CPU” and find the code below:

```
#if 0
#define  CPU_CFG_NVIC_PRIO_BITS                          4u
#endif
```

Snippet 1. CPU_CFG_NVIC_PRIO_BITS undefined

Change 0 to 1 and it will enable a CPU feature to optimize the performance of μC/OS-III:

```
#if 1
#define  CPU_CFG_NVIC_PRIO_BITS                          4u
#endif
```

Snippet 2. CPU_CFG_NVIC_PRIO_BITS defined

8. If the project is compiled now, an “include error” will occur. Open “os_cpu_c.c” file in the “uCOS_Ports” folder and find the code below:

```
#include  "../../../Source/os.h"
```

Snippet 3. The original include path

Change the code to:

```
#include  "os.h"
```

Snippet 4. The new include path

9. The last step is to change the mode of library dependency finder because these three folders are interdependent. Add the option below to the "platformio.ini" file:

```
lib_ldf_mode = deep+
```

Snippet 5. The option of Library Dependency Finder

10. Create a new assembly file in "src" in the project and copy the code below to the file:

```
    .global PendSV_Handler
    .global SysTick_Handler

    .extern OS_CPU_PendSVHandler
    .extern OS_CPU_SysTickHandler

    .text
    .align 2
    .thumb
    .syntax unified

.thumb_func
PendSV_Handler:
    b OS_CPU_PendSVHandler
    b .

.thumb_func
SysTick_Handler:
    b OS_CPU_SysTickHandler
    b .

    .end
```

Snippet 6. Assembly code for porting

The code will replace "PendSV_Handler" and "Systick_Handler" in the startup file with "OS_CPU_PendSVHandler" and "OS_CPU_SysTick-Handler" of µC/OS-III to enable context switching and multitasking.

4.1 µC/OS-III File Analysis

It will be rather easy to make a µC/OS-III based project once it is known which files in the library are supposed to serve what specific purpose. Figure 2 below is a typical architecture of a µC/OS-III application and its relationship with hardware such as CPU, Timer, and Interrupt Controller but in the exercises of the thesis, with regard to simplifying the learning environment for students, this application architecture was not adopted and minimal necessary files of µC/OS-III were selected. However, the categories and explanation of these files will still help students to better understand µC/OS-III.

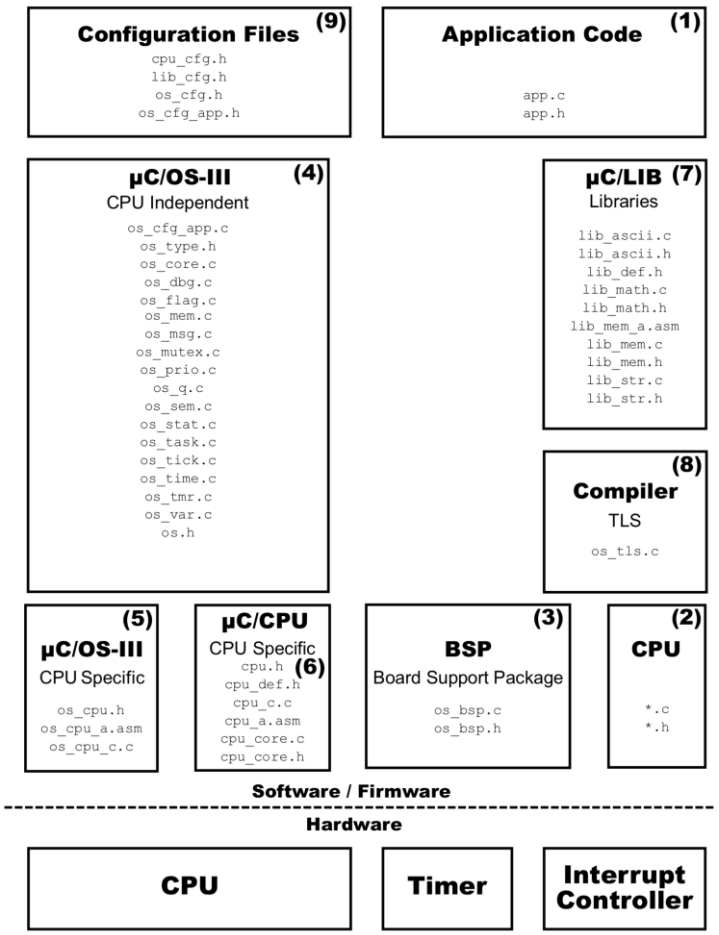

Figure 2. The typical architecture of a µC/OS-III project [1]

The directories and files shown in the figure are explained below:

(1) Application Code is where the programmer put his own source code of the project. The system will find “main()” function from here.
(2) This module usually contains source files provided by the CPU or MCU manufacturers to allow easy accesses to the peripherals of their product.
(3) The BSP is related with the specific development board used in the project and it usually provides the libraries to manipulate the peripherals of the board. In the thesis, the BSP provides the methods to use LED and LCD on the 32F429IDISCOVERY.
(4) These are the source files of μC/OS-III and all services or functions provided by μC/OS-III are included so it is where students will explore a lot when they are doing the exercises.
(5) These are so called “porting” files. They are created specifically for different CPU architectures and for different compilers.
(6) These files are heavily linked to the CPU architectures. Their purpose is to encapsulate the functions of the CPU.
(7) This folder contains some rewritten C libraries with aim of a high portability of μC/OS-III and in this case, only a single file about data type is included in the exercises because some μC/OS-III source files use it.
(8) This is an optional module about Thread-local Storage that is not needed in this thesis.
(9) The configuration files are used to tailor a project by enabling or disabling the modules of μC/OS-III. Usually most common modules in the exercises are enabled as they are not real projects and the goal is not to achieve a good performance.

4.1.1 The file “os_cpu_c.c”

This file contains some “hook” functions that were not used in the thesis and four essential functions below:

1. OSTaskStkInit: Initialize the stack frame of the task being created
2. OS_CPU_SysTickHandler: Handle the SysTick exception
3. OS_CPU_SysTickInitFreq: Initialize the SysTick using the CPU clock frequency
4. OS_CPU_SysTickInit: Initialize the SysTick using the number of counts between two ticks.

The function “OS_CPU_SysTickHandler” needs to be fully understood because it is literally the core of the entire system. The following pseudocode illustrates its tasks:

1. Enter Interrupt
2. Increments ISR nesting level
3. Increases the number of ticks by 1
4. Update the list which contains task that are delayed or pending with a timeout
5. Decrements ISR nesting level
6. Leave interrupt and trigger context switching

4.1.2 The file “os_cpu_a.s”

In this file, three are three vital functions:

1. OSStartHighRdy: Trigger the first context switching and start multitasking
2. OSCtwSw/OSIntCtwSw: both trigger a pendSV exception
3. OS_CPU_PendSVHandler: perform the work of context switching

The function “OS_CPU_PendSVHandler” is another crucial function that needs to be entirely understood. The following explanations will illustrate why it is needed:

1. If there are two tasks, Task A and B and the context switching is trigged by SysTick exception, then the kernel will switch from one task to the another at each tick. Figure 3 below shows the situation.

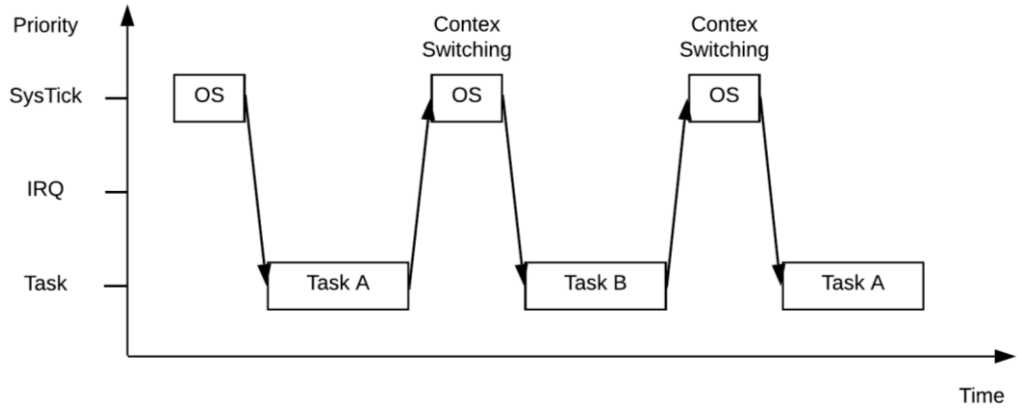

Figure 3. Context Switching by SysTick

2. If an interrupt occurs at a point and the interrupt service routine (ISR) is pre-empted by the SysTick, the ISR will be delayed after the context switching and that is unacceptable in a RTOS. Figure 4 below illustrates the situation.

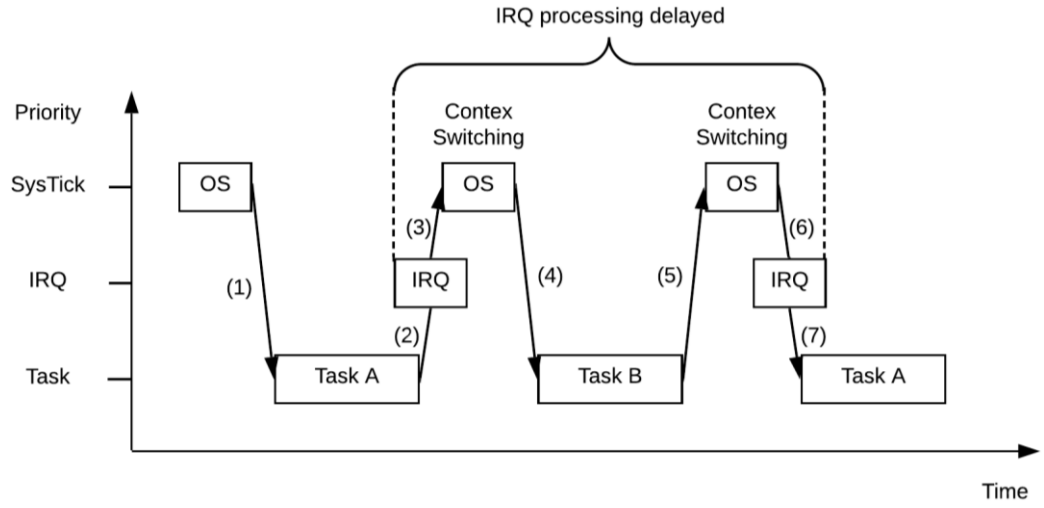

Figure 4. IRQ processing delayed

(1) The Operating System (OS) decides Task A to run next.

(2) An interrupt occurs when Task A is running.

(3) The SysTick exception preempts the ISR because it has a higher priority.

(4) After a context switching, the stack of Task A and the ISR are saved and Task B will run next.

(5) Task B is preempted by the SysTick exception.

(6) After a context switching, the stack of Task A and the ISR are reloaded and the ISR resumes but it is unacceptable to delay an interrupt processing.

(7) Task A will run next after the ISR.

3. To fix this problem, most early operating systems will detect if any interrupt is active before the context switching and only perform it when no interrupt is active. However, it is quite inefficient because if an interrupt occurs regularly at a frequency which is close to the SysTick frequency, then the context switching will never be performed.

   The use of PendSV is a much better solution. It will automatically pend the context switching until all active interrupts are completed, which is why the priority of the PendSV needs to be set at the lowest. Figure 5 below illustrates the situation.

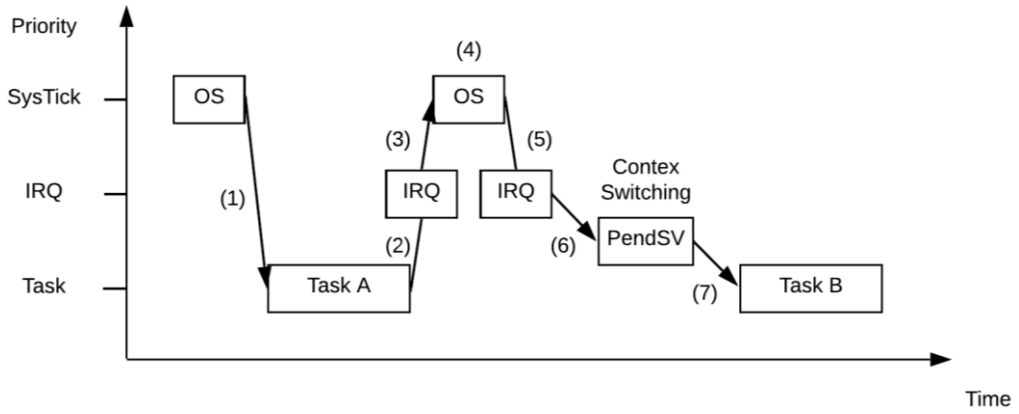

Figure 5. Context Switching by PendSV

(1) The Operating System (OS) decides Task A to run next.

(2) An interrupt occurs when Task A is running.

(3) The SysTick exception preempts the interrupt because it has a higher priority.

(4) The OS triggers the PendSV exception and does not perform context switching.
(5) The interrupt service routine (ISR) completes its job.
(6) The PendSV handler performs context switching.
(7) Task B will run next after the switching.

4.2 μC/OS-III Application Startup

The following figure demonstrates the startup process of a μC/OS-III application:

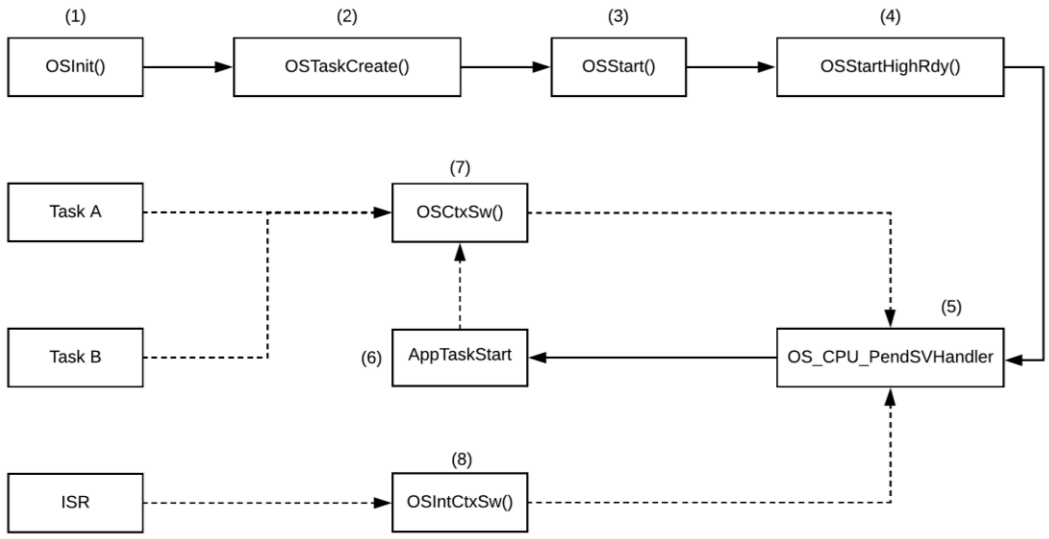

Figure 6. The startup of a μC/OS-III application

(1) Initialize the Operating System
(2) Create the first task "AppTaskStart" which is going to create other tasks
(3) Mark the highest priority task and set the status of system to "running"
(4) Set the priority of PendSV handler to the lowest and enable it and load the process stack pointer of the task "AppTaskStart"
(5) Perform the first context switching to start the task "AppTaskStart"
(6) Initialize the system clock and all peripherals and create two tasks: "Task A" and "Task B"

(7) All tasks will call the function “OSCtrSw()” sequentially to trigger PendSV exception for context switching.

(8) When the interrupt service routine (ISR) is completed, it will call the function “OSIntCtxSw()” to trigger PendSV exception for context switching.

5 EXERCISE DESIGN

5.1 Preemptive Scheduling

5.1.1 Background Theory

The preemptive scheduling works in a way that an OS (Operating System) will allocate a specific CPU time duration to a task for running but the duration is probably not enough for the task to complete its job. After the duration, the OS will not consider if the job is finished or not and selects the highest priority task in the ready status to run. It solves a problem in the non-preemptive scheduling that other tasks may wait for a long time for a task to finish its job and, which is not acceptable in the RTOS.

5.1.2 Instruction

As it is the first exercise, students will be first told to download and run the source code of this exercise. Then they will monitor what this program does and learn why it does in this way by studying the basic functions of μC/OS-III, such as creating tasks and time delay. After that, they will be told to comment the delay in the LED3 task and LED4 task respectfully and explain what happens with the concept of preemptive scheduling.

5.1.3 Result Analysis

The result is at the beginning that both LEDs toggle at 1- second interval and once the delay in the LED3 task is commented, LED3 will always be on while LED4 never lights up but when the delay in the LED4 task is commented, LED3 still toggles at 1- second interval and LED4 is on forever.

1. The diagram below illustrates the condition when the delay in LED3 is commented.

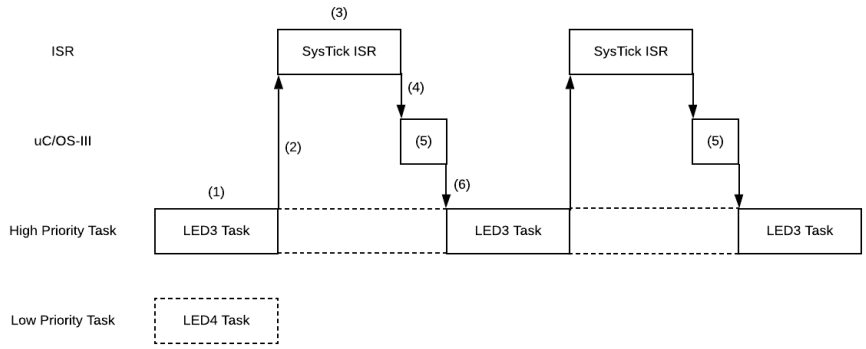

Figure 7. LED3 Delay commented

(1) LED3 task is running at the system frequency because no time delay inside and the system timer (SysTick) interrupt occurs.
(2) If interrupts are enabled, CPU jumps to the SysTick Interrupt Service Routine (ISR).
(3) SysTick ISR is doing its job and increases the system ticks by one.
(4) SysTick ISR completed and CPU turns back to uC/OS-III.
(5) uC/OS-III checks the Ready List (where tasks that are ready to run are placed) and selects a task with the highest priority to run.
(6) LED3 task is going to run again because its priority is higher than LED4 task though both them are in the Ready List.

2. The diagram below illustrates the condition when the delay in LED4 is commented.

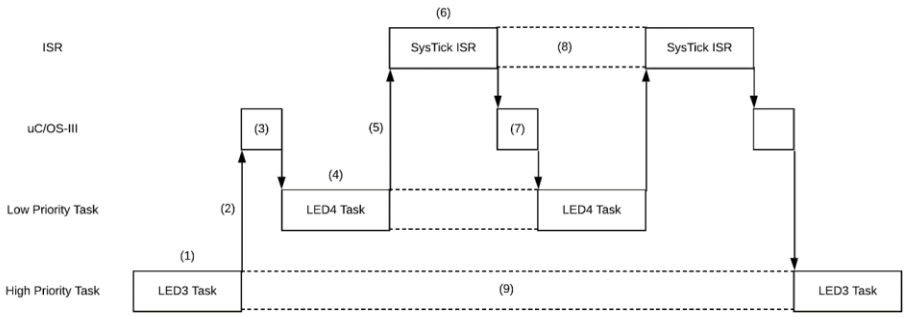

Figure 8. LED4 Delay commented

(1) LED3 task as a high priority task is running first.
(2) Time delay occurs and CPU goes to uC/OS-III for scheduling.
(3) uC/OS-III checks the Ready List and makes LED4 task to run.

(4) LED4 is running at the system frequency and SysTick interrupt occurs.
(5) CPU goes to SysTick ISR.
(6) SysTick ISR is doing its job and increases the system ticks by one.
(7) uC/OS-III makes LED4 task to run again because LED3 task is still waiting for the delay to pass and LED4 task is the only available task in the Ready List.
(8) SysTick ISR will run 1000 times if the delay of LED3 task is 1s and the system frequency is 1ms.
(9) LED3 task will run again after the delay passed.

5.2 Task Synchronization

5.2.1 Background Theory

In a RTOS, a popular way to carry out task synchronization is to use a semaphore. A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. When it is used to protect a share resource, it is a key and the task who wants to access the resource must get the key first and when it is used for synchronization, it is a signal sent from one task to another that allows it to run.

In μC/OS-III, a task semaphore is a semaphore that is simply built into a task so when a task (Task A) wants to synchronize another task (Task B) and it knows exactly the Task Control Block (TCB) of Task B, then it can easily use the task semaphore of Task B instead of creating a global semaphore.

5.2.2 Instructions

Students will first study the Application Programming Interface (API) of the semaphore and task semaphore, such as the creation, post and pend. There are two levels in this exercise. For the basic one, students will be asked to synchronize LED3 with LED4 by a normal global semaphore and for the advanced level, they will try to synchronize one more task, an UART task, by their task semaphores.

5.2.3 Result Analysis

The result of the basic level is that LED4 toggles right after LED3 while only the LED3 task has the delay. The diagram (Figure 9) below illustrates the process.

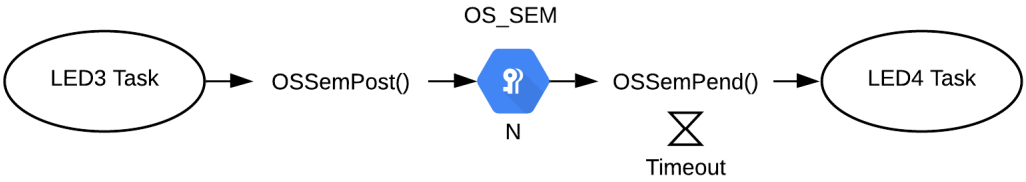

Figure 9. Basic level synchronization

The result of the advanced level is that LED4 toggles following LED3 and is followed by the UART task, which sends a message to PC. The diagram (Figure 10) below illustrates the process.

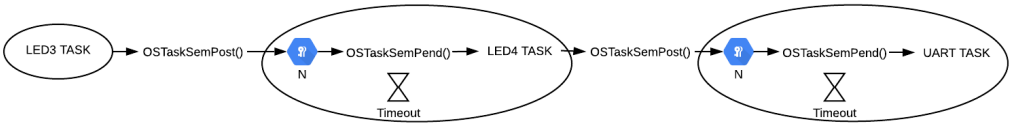

Figure 10. Advanced level synchronization

5.3 Multiple Tasks Synchronization

5.3.1 Background Theory

Event flags are usually used to synchronize multiple tasks or interrupts. A task can be synchronized with all executions of other tasks, which is an “AND” condition or it can be synchronized with any single execution of them, which is an “OR” condition. To use event flags, an Event Flag Group must be created first, which is an 8-, 16- or 32-bits data type, and when a task wants to post an event flag to this group, it literally sets one bit of this data type to 1.

5.3.2 Instructions

The students will study the API of event flag first and then they will be asked to synchronize the UART with LED3 and LED4 in two conditions. The first condition is “AND” which means the UART task runs after both LED3 and LED4 execute and the second condition is “or”, which means UART can run after either of LED3 or LED4 executes.

5.3.3 Result Analysis

In “AND” condition, the UART will send a message to PC after both LEDs toggle and in “or” condition, the UART will send a message to PC after either of LEDs toggles. The diagram below (Figure 11) illustrates the process.

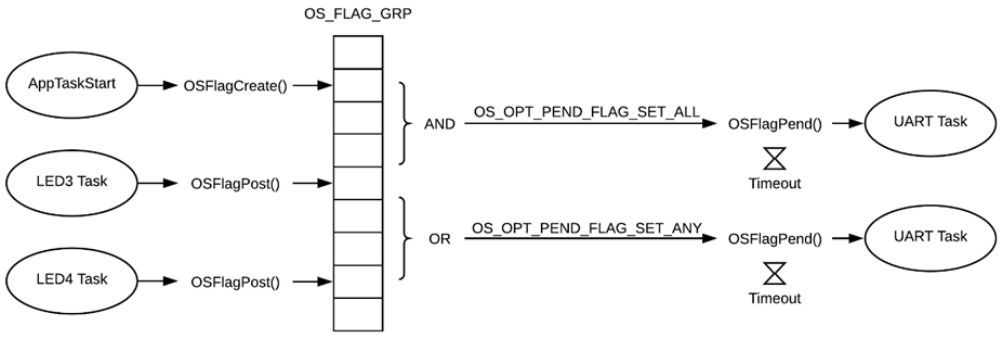

Figure 11. Event Flags exercise

5.4 Priority Inversion

5.4.1 Background Theory

Priority inversion is a situation where a high priority task is waiting for the release of a shared resource held by a low priority task while the low priority task is preempted by another task. Hence, the high priority task supposed to run first will run last after obtaining the shared resource, which is not tolerable in a RTOS.

The priority inversion usually occurs when a normal semaphore is used to protect shared resources. With the intention of solving this problem, Mutual Exclusion (Mutex) semaphore is a better alternative to the normal one. A Mutex semaphore is a special type of binary semaphore and when a high priority task wants a shared resource held by a low priority task, the Mutex semaphore will temporarily lift the priority of the low priority task to the highest and makes it release the shared resource as soon as possible. After releasing the resource, the Mutex semaphore will lower its priority and the high priority task starts processing on the resource.

The diagram below (Figure 12) illustrates the situation of priority inversion:

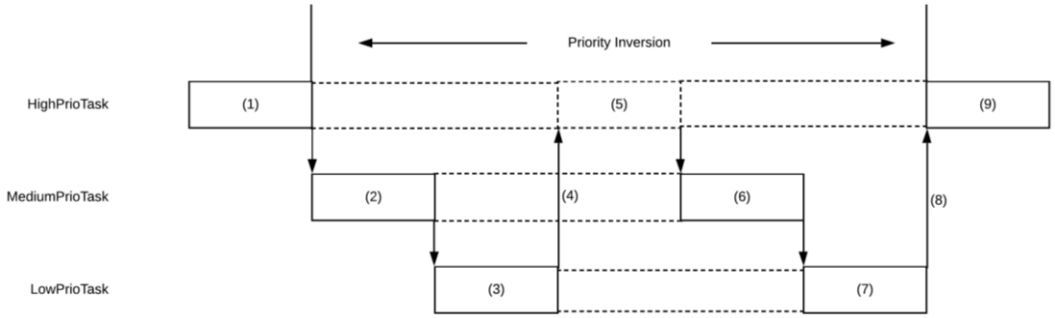

Figure 12. Priority Inversion

(1) HighPrioTask acquires a semaphore, runs and releases the semaphore.
(2) MediumPrioTask is running.
(3) LowPrioTask acquires the semaphore and is running.
(4) HighPrioTask preempts LowPrioTask.
(5) HighPrioTask does not get the semaphore as it is still held by LowPrioTask.
(6) MediumPrioTask is running.
(7) MediumPrioTask done and LowPrioTask continues running.
(8) LowPrioTask done and releases the semaphore.
(9) HighPrioTask finally receives the semaphore and continue its job.

The diagram below (Figure 13) illustrates the situation using Mutex semaphore.

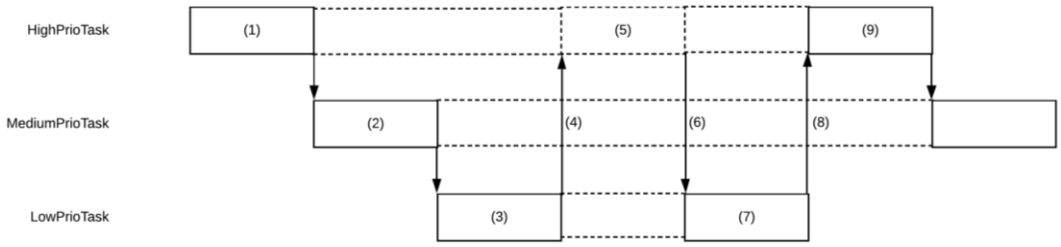

Figure 13. Mutex semaphore

(1) HighPrioTask acquires a Mutex semaphore, runs and releases the Mutex semaphore.
(2) MediumPrioTask is running.
(3) LowPrioTask acquires the Mutex semaphore and is running.
(4) HighPrioTask preempts LowPrioTask.
(5) HighPrioTask does not get the Mutex semaphore as it is still held by LowPrioTask.
(6) μC/OS-III raises the priority of LowPrioTask.
(7) LowPrioTask resumes and releases the Mutex semaphore after completed its job.
(8) μC/OS-III lowers the priority of LowPrioTask.
(9) HighPrioTask receives the semaphore and continues its job.

5.4.2 Instructions

The aim of this exercise is to help students to better understand priority inversion caused by using a normal semaphore and compare it with the use of a Mutex semaphore. They still need to study the API of Mutex semaphore first and study the source code of this exercise. After they understand it, they can simply run it and check the UART simulations. The “for loop” below in the source code needs to be paid more attention because inside it, the situation is simulated that a shared resource is being held by a low priority task while scheduling takes place:

```
for (sharedInt = 0; sharedInt < 300000; sharedInt++)
{
    OSSched();
}
```

Snippet 7. The for loop in Priority Inversion

5.4.3 Result Analysis

The screenshot below (Figure 14) is the simulation of priority inversion and it shows that the HighPrioTask is waiting for the release of the resource from the LowPrioTask.

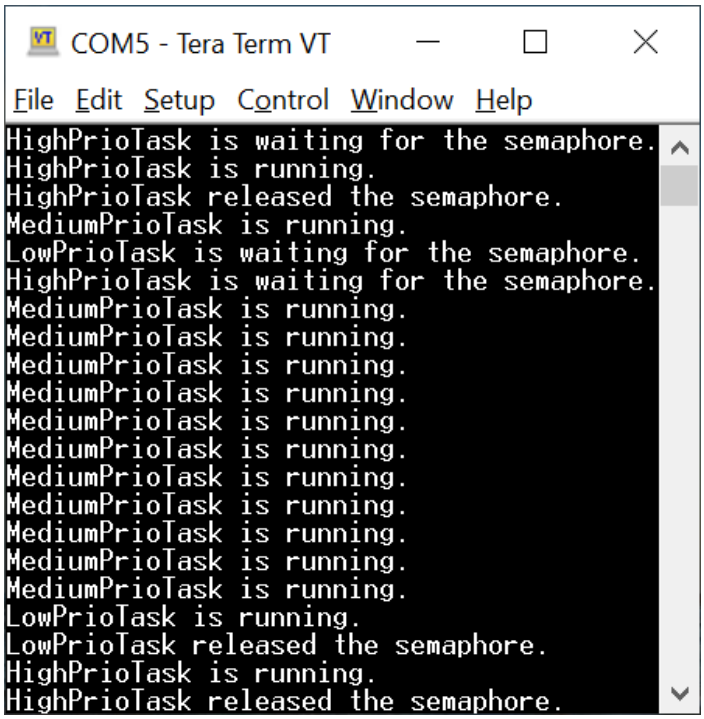

Figure 14. Priority Inversion simulation

The screenshot below (Figure 15) is the simulation of using Mutex semaphore demonstrating that whenever the HighPrioTask is waiting for the resource, the LowPrioTask will run next and releases the resource once it completes its job.

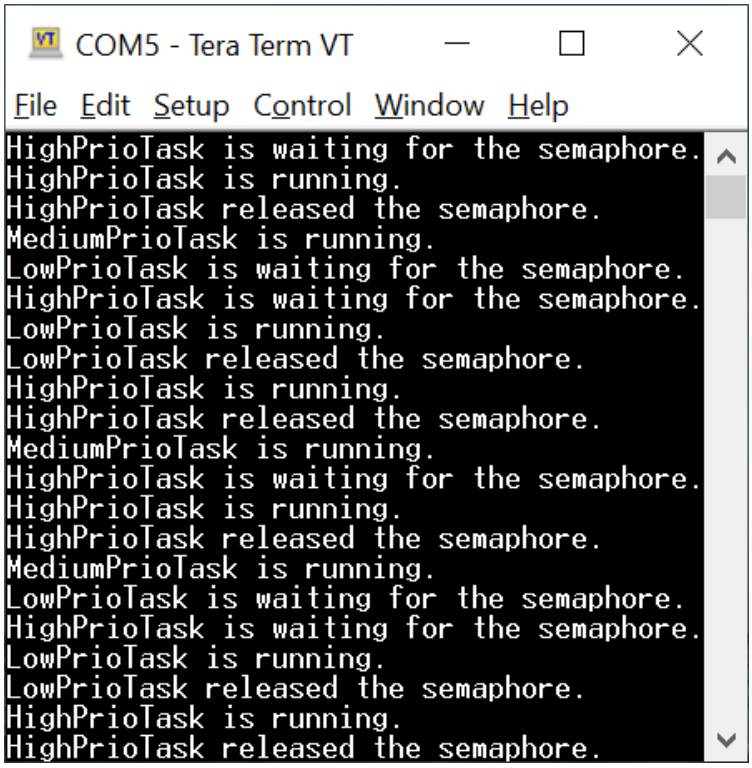

Figure 15. Mutex semaphore simulation

5.5 Serial Port Echo

5.5.1 Background Theory

A message consists of a pointer to data, a variable containing the size of the data being pointed to, and a timestamp indicating when the message was sent. A message queue is a kernel object that is responsible for holding messages and a task message queue is the message queue built into a task. It is easy to use task message queue if the sender knows exactly to where messages are sent.

5.5.2 Instructions

In this exercise, an UART interrupt is used to receive a message of unknown size and then posts this message to a task message queue. Meanwhile, a task is pending for the message in its task message queue and after receiving it, the task will send

it back to PC by UART. The initialization function and the function to achieve an unknown size transmission will be provided to students.

5.5.3 Result Analysis

The result is that no matter how long the UART message sent through a serial communication tool is and even it is a file, it will immediately show up on the serial monitor. The following source code snippets will illustrate the process:

1. Active UART interrupt in the first task:

```
__HAL_UART_ENABLE_IT(&huart1, UART_IT_IDLE);
HAL_UART_Receive_IT(&huart1, rxData, MAX_SIZE);
```

Snippet 8. UART IDLE interrupt activation

2. Post the message received by the interrupt to task message queue:

```
void USART1_IRQHandler(void)
{
    OS_ERR err;

    uint32_t isrflags = READ_REG(huart1.Instance->SR);
    uint32_t cr1its = READ_REG(huart1.Instance->CR1);

    /* IDLE interrupt detection and handler to achieve unknown size transmission */
    if (((isrflags & USART_SR_IDLE) != RESET) && ((cr1its & USART_CR1_IDLEIE)))
    {
        __HAL_UART_CLEAR_IDLEFLAG(&huart1);
        huart1.RxState = HAL_UART_STATE_READY;

        rxLen = MAX_SIZE - huart1.RxXferCount;
        OSTaskQPost((OS_TCB *)&UartTransmitTaskTCB,
                    (void *)rxData,
                    (OS_MSG_SIZE)rxLen,
                    (OS_OPT)OS_OPT_POST_FIFO,
                    (OS_ERR *)&err);
    }

    HAL_UART_IRQHandler(&huart1);
}
```

Snippet 9. UART ISR source code

3. Whenever the task receives the message from its task message queue, it sends the message back and activates the interrupt again:

```
while (DEF_TRUE)
{
    txData = OSTaskQPend((OS_TICK)0, //Wait for messages from UART interrupt
                         (OS_OPT)OS_OPT_PEND_BLOCKING,
                         (OS_MSG_SIZE *)&msg_size,
                         (CPU_TS *)&ts,
                         (OS_ERR *)&err);

    HAL_UART_Transmit_IT(&huart1, txData, msg_size); //Transmit back
    HAL_UART_Receive_IT(&huart1, rxData, MAX_SIZE);  //Make ready for next interrupt
    BSP_LED_Toggle(LED4);
}
```

Snippet 10. Transmit Task source code

5.6 Task Communication

5.6.1 Background Theory

There is a mechanism called “bilateral rendez-vous” by which two tasks can synchronize their activities by using their task message queues. It can accomplish the same performance of the synchronization by using a semaphore and moreover, both tasks can also send messages to each other.

5.6.2 Instructions

Students will be asked to create two tasks Task A and Task B and add a counter to both tasks that will count how many times the task has been run. Using the mechanism mentioned in Background Theory, students then need to make each task to print the running times of another task by UART.

5.6.3 Result Analysis

The picture below (Figure 16) is the result on a serial monitor.

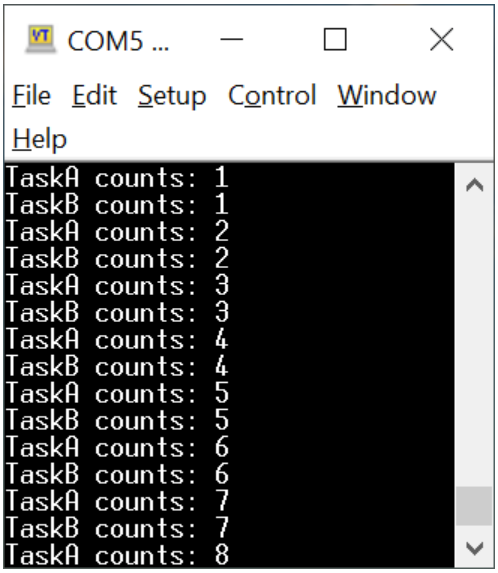

Figure 16. Serial print of Task Communication

The diagram below (Figure 17) illustrates the process.

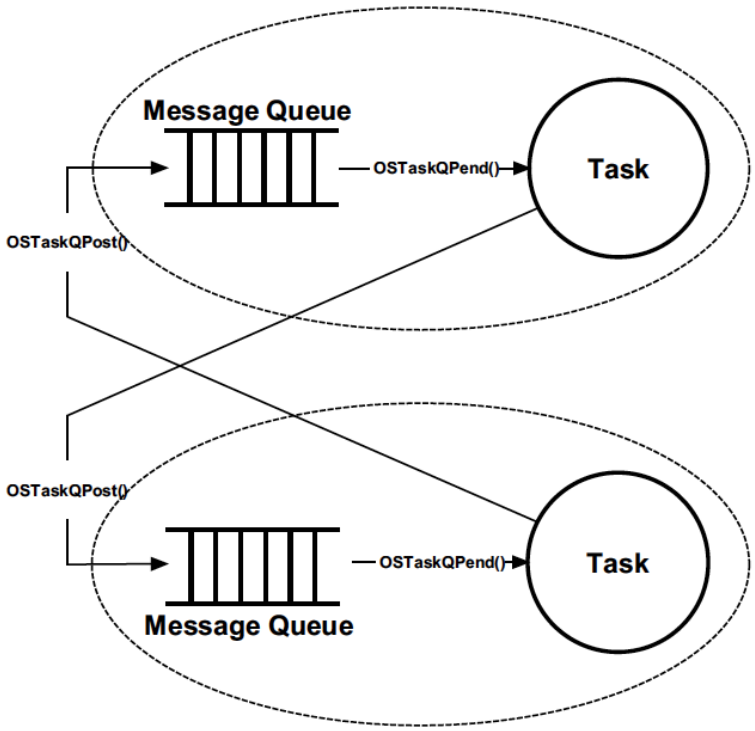

Figure 17. Task Message Queue synchronization

5.7 Flow Control

5.7.1 Background Theory

This exercise is the last exercise and the most difficult one. It combines the use of semaphore, task semaphore and task message queue. Students so far have learnt their APIs from the previous exercises, so this exercise will be an appropriate practice for them.

5.7.2 Instruction

The exercise is to display a file sent through UART from PC on the LCD. However, the screen size of the LCD is limited so it cannot display the whole file at once, it is then necessary to slice the file into many messages and only display each message on the LCD for a while. Students will be told to have an UART interrupt to receive the file, a task that is going to slice the file and another task whose mission is to display the file on the LCD but they need to figure out by themselves how to synchronize the activities of these two tasks and an interrupt by using semaphore, task semaphore and task message queue. All initialization functions of UART and System clock will be provided.

5.7.3 Result Analysis

With the reference to the source code (Appendix 1), the diagram below (Figure 18) illustrates the process.

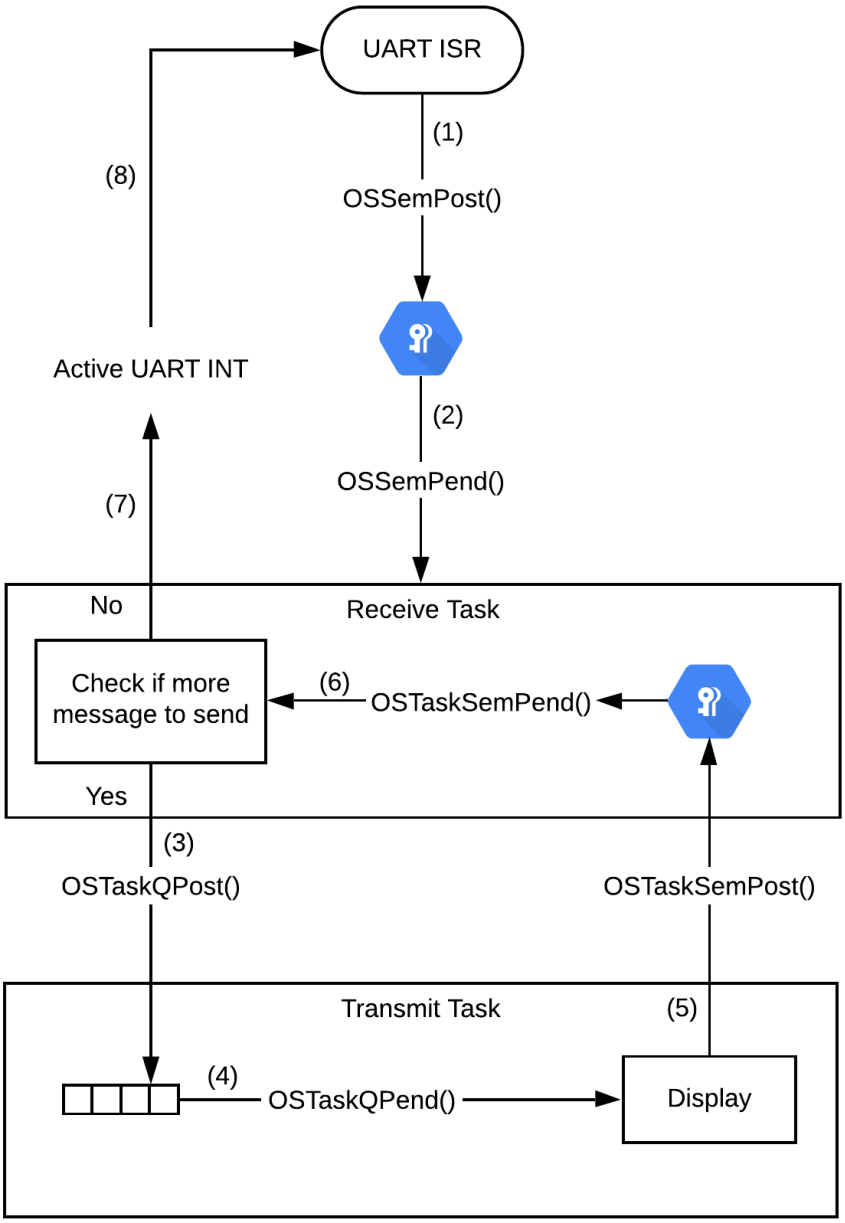

Figure 18. Flow Control process

(1) UART ISR posts a semaphore to the Receive Task after receiving a file

(2) Receive Task receives the semaphore and knows a file is received

(3) Receive Task posts the first piece of the file into the task message queue of Transmit Task.

(4) Transmit Task receives the first message and displays it.
(5) Transmit Task indicates display completed by posting 1 into the task semaphore of Receive Task.
(6) Receive Task checks if there are more pieces of the file to send.
(7) If no more messages to send, Receive Task will enable UART interrupt again.
(8) UART interrupt receives the next file.

6 TESTING

The testing was done by the designer who cleaned the whole environment first and reimplemented everything again by following the instructions written in the previous chapters. There might be a situation that a step is missed in the instructions but was did intuitively so this testing is not truly objective. Anyway, it turns out that a test for the instruction is necessary because there is a configuration file missed and the library failed to work at the end. In addition, one major problem found during the test is the high complexity of making μC/OS-III library. The miss of a single file can result a broken or unstable system, although all three folders can be first sorted out by the lecturer and provided to students directly. To fix this problem, one possible solution is that someone registers the library on PlatformIO and students then can easily install it from a remote repository but this method requires the study of two "json" files. Another problem is the short of introduction of PlatformIO utilities, such as its built-in serial monitor and its PIO Unified Debugger. In the instructions above, only the library options in the file "platformio.ini" are mentioned but in fact, from this file someone can not only set up the serial monitor, but also control building and uploading processes. The third problem is that instruction of some exercises may not be clear enough for a student who is totally new to μC/OS-III. For example, the recommended way to initialize and start the system is not instructed so if a student programs CPU and hardware configurations before the initialization of the OS, or creates multiple tasks before the start of the OS, the program will probably not behave as expected. Meanwhile, some exercises can also be done with different methods so the instructions should not restrict strictly the objects students use as long as they are based on μC/OS-III.

7 CONCLUSIONS

In summary, a new RTOS teaching environment was developed in the thesis, based on the RTOS kernel μC/OS-III, the development board 32F429IDISCOVERY and a programming IDE PlatformIO. In addition, ten exercises were designed covering the subjects of Scheduling, Synchronization, Resource Management and Message Passing.

Nevertheless, there is still a big room for further improvement. First, the PlatfromIO IDE is not integrated with STM32CubeMX so the development of any STM32 boards using STM32Cube framework is not truly seamless and convenient. If students want to explore more of the board and use STM32CubeMX, then they probably need to install a Python script by which they can convert the project generated by STM32CubeMX to a PlatformIO project. Besides that, not all exercises are sophisticated. For example, two similar exercises of semaphore and task semaphore or message queue and task message queue can be merged into one exercise and the topics of Interrupt Management, Timer Management or Run-Time Statistics should also be paid attention.

Another idea of making the course even better is to make an integration between other courses. The introduced development board and environment in this thesis can also be used in other courses, such as Embedded System Design or Project. In the Design course, students are encouraged to explore as many peripherals as they want while in the RTOS course, they will save more time for the learning of RTOS. Finally, in the Project course, they can combine what they learned from two previous courses and make a big project based on RTOS. Then these three courses will compose a series of courses.

REFERENCES

[1] J. Royal, "uC/OS-III Documentation," 03 October 2013. [Online]. Available: https://doc.micrium.com/pages/viewpage.action?pageId=10753180. Accessed 21 May 2020.

[2] ARM, ARM Cortex-M4 Processor Technical Reference Manual, Cambridge, 2015.

[3] STMicroelectronics, UM1670 User manual, Geneva: STMicroelectronics, 2017.

[4] I. Kravets, "platformio doc," 8 April 2020. [Online]. Available: https://docs.platformio.org/en/latest/. Accessed 21 May 2020.

```
1.  /*
2.  *********************************************************************
    *********************************
3.  *                                                 LOCAL INCLUDES
4.  *********************************************************************
    *********************************
5.  */
6.
7.  #include "stm32f4xx_hal.h"
8.  #include "stm32f429i_discovery_lcd.h"
9.  #include "os.h"
10. #include "string.h"
11.
12. /*
13. *********************************************************************
    *********************************
14. *                                                 LOCAL DEFINES
15. *********************************************************************
    *********************************
16. */
17.
18. /* Task Stack Size */
19. #define APP_TASK_START_STK_SIZE 256u
20. #define UART_TASK_STK_SIZE 256u
21.
22. /* Task Priority */
23. #define APP_TASK_START_PRIO 1u
24. #define UART_RECEIVE_TASK_PRIO 12u
25. #define UART_TRANSMIT_TASK_PRIO 22u
26.
27. /* UART and LCD Display */
28. #define FILE_SIZE 1047u     //need to know the file size before trans-
    mission
29. #define MAX_COLUMNS 14u
30. #define MAX_ROWS 12
31.
32. /*
33. *********************************************************************
    *********************************
34. *                                               GLOBAL VARIABLES
35. *********************************************************************
    *********************************
36. */
37.
38. /* Task Control Block */
39. static OS_TCB AppTaskStartTCB;
40. static OS_TCB UartTransmitTaskTCB;
41. static OS_TCB UartReceiveTaskTCB;
42.
43. /* Task Stack */
44. static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];
45. static CPU_STK UartTransmitTaskStk[UART_TASK_STK_SIZE];
46. static CPU_STK UartReceiveTaskStk[UART_TASK_STK_SIZE];
47.
48. /* OS Kernal Objects */
49. OS_SEM rxSem;
50.
51. /* UART */
52. UART_HandleTypeDef huart1;
53. uint8_t rxData[FILE_SIZE];
54.
55. /*
```

```
56. ********************************************************************************
57. *                                      FUNCTION PROTOTYPES
58. ********************************************************************************
59. */
60. 
61. /* Task Prototypes */
62. static void AppTaskStart(void *p_arg);
63. static void UartTransmitTask(void *p_arg);
64. static void UartReceiveTask(void *p_arg);
65. 
66. /* System Initilization Prototypes */
67. void SystemClock_Config(void);
68. static void MX_USART1_UART_Init(void);
69. void LCD_Init(void);
70. 
71. /*
72. ********************************************************************************
73. *                                                MAIN
74. ********************************************************************************
75. */
76. 
77. int main(void)
78. {
79.     OS_ERR err;
80. 
81.     OSInit(&err);
82. 
83.     OSSemCreate((OS_SEM *)&rxSem,
84.                 (CPU_CHAR *)"Synchronization Semaphore",
85.                 (OS_SEM_CTR)0,
86.                 (OS_ERR *)&err);
87. 
88.     OSTaskCreate((OS_TCB *)&AppTaskStartTCB,
89.                  (CPU_CHAR *)"App Task Start",
90.                  (OS_TASK_PTR)AppTaskStart,
91.                  (void *)0,
92.                  (OS_PRIO)APP_TASK_START_PRIO,
93.                  (CPU_STK *)&AppTaskStartStk[0],
94.                  (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
95.                  (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
96.                  (OS_MSG_QTY)5u,
97.                  (OS_TICK)0u,
98.                  (void *)0,
99.                  (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
100.                        (OS_ERR *)&err);
101. 
102.           OSStart(&err);
103.       }
104. 
105.       /*
106.       ********************************************************************************
    ************************************
107.       *                                                STARTUP TASK
108.       ********************************************************************************
    ************************************
109.       */
110. 
111.       static void AppTaskStart(void *p_arg)
112.       {
```

```
113.            OS_ERR err;
114.
115.            HAL_Init();
116.
117.            SystemClock_Config();
118.
119.            BSP_LED_Init(LED3);
120.            BSP_LED_Init(LED4);
121.
122.            MX_USART1_UART_Init();
123.            HAL_UART_Receive_IT(&huart1, rxData, FILE_SIZE);
124.
125.            LCD_Init();
126.            BSP_LCD_DisplayStrin-
    gAtLine(1, (uint8_t *)"LCD_Init Done");        //Indicate success-
    ful init
127.
128.            OSTaskCreate((OS_TCB *)&UartReceiveTaskTCB,
129.                         (CPU_CHAR *)"Uart Receive Task",
130.                         (OS_TASK_PTR)UartReceiveTask,
131.                         (void *)0,
132.                         (OS_PRIO)UART_RECEIVE_TASK_PRIO,
133.                         (CPU_STK *)&UartReceiveTaskStk[0],
134.                         (CPU_STK_SIZE)UART_TASK_STK_SIZE / 10,
135.                         (CPU_STK_SIZE)UART_TASK_STK_SIZE,
136.                         (OS_MSG_QTY)5u,
137.                         (OS_TICK)0u,
138.                         (void *)0,
139.                         (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_C
    LR),
140.                         (OS_ERR *)&err);
141.
142.            OSTaskCreate((OS_TCB *)&UartTransmitTaskTCB,
143.                         (CPU_CHAR *)"Uart Transmit Task",
144.                         (OS_TASK_PTR)UartTransmitTask,
145.                         (void *)0,
146.                         (OS_PRIO)UART_TRANSMIT_TASK_PRIO,
147.                         (CPU_STK *)&UartTransmitTaskStk[0],
148.                         (CPU_STK_SIZE)UART_TASK_STK_SIZE / 10,
149.                         (CPU_STK_SIZE)UART_TASK_STK_SIZE,
150.                         (OS_MSG_QTY)5u,
151.                         (OS_TICK)0u,
152.                         (void *)0,
153.                         (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_C
    LR),
154.                         (OS_ERR *)&err);
155.        }
156.
157.        /*
158.        ***************************************************************
    ***************************************
159.        *                                                      TASKS
160.        ***************************************************************
    ***************************************
161.        */
162.
163.        static void UartReceiveTask(void *p_arg)
164.        {
165.            OS_ERR err;
166.            CPU_TS ts;
167.
168.            while (DEF_TRUE)
169.            {
```

```
170.                OSSemPend((OS_SEM *)&rxSem,     //Wait for a sema-
    phore sent by UART interrupt
171.                          (OS_TICK)0,
172.                          (OS_OPT)OS_OPT_PEND_BLOCKING,
173.                          (CPU_TS *)&ts,
174.                          (OS_ERR *)&err);
175.
176.                for (CPU_INT32U length = 0; length < FILE_SIZE; length +=
     MAX_COLUMNS)      //Slice whole received data into message queues
177.                {
178.                    OSTaskQPost((OS_TCB *)&UartTransmit-
    TaskTCB,     //Send a message to transmit task
179.                                (void *)&rxData[length],
180.                                (OS_MSG_SIZE)14,
181.                                (OS_OPT)OS_OPT_POST_FIFO,
182.                                (OS_ERR *)&err);
183.
184.                    OSTaskSemPend((OS_TICK)0,       //Wait for a notifi-
    cation to send next message
185.                                  (OS_OPT)OS_OPT_PEND_BLOCKING,
186.                                  (CPU_TS *)&ts,
187.                                  (OS_ERR *)&err);
188.                }
189.                BSP_LED_Toggle(LED4);       //Indicate completed dis-
    play
190.                HAL_UART_Re-
    ceive_IT(&huart1, rxData, FILE_SIZE);        //Activate UART inter-
    rupt again
191.            }
192.        }
193.
194.        static void UartTransmitTask(void *p_arg)
195.        {
196.            OS_ERR err;
197.            CPU_INT08U *txData;
198.            OS_MSG_SIZE msg_size;
199.            CPU_TS ts;
200.            CPU_INT08U line = 0;
201.
202.            while (DEF_TRUE)
203.            {
204.                txData = OSTaskQPend((OS_TICK)0,        //Wait for a mes-
    sage from receive task
205.                                     (OS_OPT)OS_OPT_PEND_BLOCKING,
206.                                     (OS_MSG_SIZE *)&msg_size,
207.                                     (CPU_TS *)&ts,
208.                                     (OS_ERR *)&err);
209.
210.                if (++line > MAX_ROWS)      //Decide which line to dis-
    play
211.                {
212.                    BSP_LCD_Clear(LCD_COLOR_WHITE);
213.                    line = 1;
214.                }
215.
216.                BSP_LCD_DisplayStringAtLine(line, txData);      //Dis-
    play the message
217.
218.                OSTimeDlyHMSM((CPU_INT16U)0,
219.                              (CPU_INT16U)0,
220.                              (CPU_INT16U)1u,
221.                              (CPU_INT32U)0,
222.                              (OS_OPT)OS_OPT_TIME_HMSM_STRICT,
```

```
223.                                (OS_ERR *)&err);
224.
225.                OSTaskSemPost((OS_TCB *)&UartReceiveTaskTCB,         //No-
    tify receive task to send next message
226.                                (OS_OPT)OS_OPT_POST_NONE,
227.                                (OS_ERR *)&err);
228.            }
229.        }
230.
231.        /*
232.        ************************************************************************
    *************************************
233.        *                                              NON-TASK FUNCTIONS
234.        ************************************************************************
    *************************************
235.        */
236.
237.        void USART1_IRQHandler(void)
238.        {
239.            HAL_UART_IRQHandler(&huart1);       //STM32 general IRQ handl
    er
240.        }
241.
242.        void HAL_UART_RxCpltCallback(UART_Hand-
    leTypeDef *huart)     //Weak function called from IRQ handler when re-
    ceive completed
243.        {
244.            OS_ERR err;
245.            OSSemPost((OS_SEM *)&rxSem,
246.                      (OS_OPT)OS_OPT_POST_1,
247.                      (OS_ERR *)&err);
248.        }
249.
250.        void HAL_Delay(uint32_t Delay)
251.        {
252.            OS_ERR err;
253.            OSTimeDly((OS_TICK)Delay,
254.                      (OS_OPT)OS_OPT_TIME_DLY,
255.                      (OS_ERR *)&err);
256.        }
257.
258.        /*
259.        ************************************************************************
    *************************************
260.        *                                              System Initializations
261.        ************************************************************************
    *************************************
262.        */
263.
264.        void LCD_Init(void)
265.        {
266.            BSP_LCD_Init();
267.            BSP_LCD_LayerDefaultInit(LCD_BACK-
    GROUND_LAYER, LCD_FRAME_BUFFER);
268.            BSP_LCD_LayerDefaultInit(LCD_FORE-
    GROUND_LAYER, LCD_FRAME_BUFFER);
269.            BSP_LCD_SelectLayer(LCD_FOREGROUND_LAYER);
270.            BSP_LCD_DisplayOn();
271.            BSP_LCD_Clear(LCD_COLOR_WHITE);
272.            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
273.        }
274.
275.        static void MX_USART1_UART_Init(void)
```

```
276.        {
277.            huart1.Instance = USART1;
278.            huart1.Init.BaudRate = 115200;
279.            huart1.Init.WordLength = UART_WORDLENGTH_8B;
280.            huart1.Init.StopBits = UART_STOPBITS_1;
281.            huart1.Init.Parity = UART_PARITY_NONE;
282.            huart1.Init.Mode = UART_MODE_TX_RX;
283.            huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
284.            huart1.Init.OverSampling = UART_OVERSAMPLING_16;
285.            if (HAL_UART_Init(&huart1) == HAL_OK)
286.            {
287.                BSP_LED_On(LED3);
288.            }
289.        }
290.
291.        void HAL_UART_MspInit(UART_HandleTypeDef *huart)
292.        {
293.            GPIO_InitTypeDef GPIO_InitStruct = {0};
294.            if (huart->Instance == USART1)
295.            {
296.                /* Peripheral clock enable */
297.                __HAL_RCC_USART1_CLK_ENABLE();
298.
299.                __HAL_RCC_GPIOA_CLK_ENABLE();
300.                /**USART1 GPIO Configuration
301.                PA9 ------> USART1_RX
302.                PA10 ------> USART1_TX
303.                */
304.                GPIO_InitStruct.Pin = GPIO_PIN_9 | GPIO_PIN_10;
305.                GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
306.                GPIO_InitStruct.Pull = GPIO_NOPULL;
307.                GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
308.                GPIO_InitStruct.Alternate = GPIO_AF7_USART1;
309.                HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
310.
311.                /* USART1 interrupt Init */
312.                HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
313.                HAL_NVIC_EnableIRQ(USART1_IRQn);
314.            }
315.        }
316.
317.        void SystemClock_Config(void)
318.        {
319.            RCC_OscInitTypeDef RCC_OscInitStruct = {0};
320.            RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
321.            RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};
322.
323.            /** Configure the main internal regulator output voltage
324.          */
325.            __HAL_RCC_PWR_CLK_ENABLE();
326.            __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLT-
    AGE_SCALE1);
327.            /** Initializes the CPU, AHB and APB busses clocks
328.          */
329.            RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
330.            RCC_OscInitStruct.HSIState = RCC_HSI_ON;
331.            RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRA-
    TION_DEFAULT;
332.            RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
333.            RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
334.            RCC_OscInitStruct.PLL.PLLM = 8;
335.            RCC_OscInitStruct.PLL.PLLN = 180;
336.            RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
```

```
337.           RCC_OscInitStruct.PLL.PLLQ = 7;
338.           HAL_RCC_OscConfig(&RCC_OscInitStruct);
339.
340.           /** Activate the Over-Drive mode
341.         */
342.           HAL_PWREx_EnableOverDrive();
343.
344.           /** Initializes the CPU, AHB and APB busses clocks
345.         */
346.           RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCK-
    TYPE_SYSCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
347.           RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
348.           RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
349.           RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
350.           RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
351.
352.           HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5);
353.
354.           PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPH-
    CLK_LTDC;
355.           PeriphClkInitStruct.PLLSAI.PLLSAIN = 216;
356.           PeriphClkInitStruct.PLLSAI.PLLSAIR = 2;
357.           PeriphClkInitStruct.PLLSAIDivR = RCC_PLLSAIDIVR_2;
358.           HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);
359.       }
```