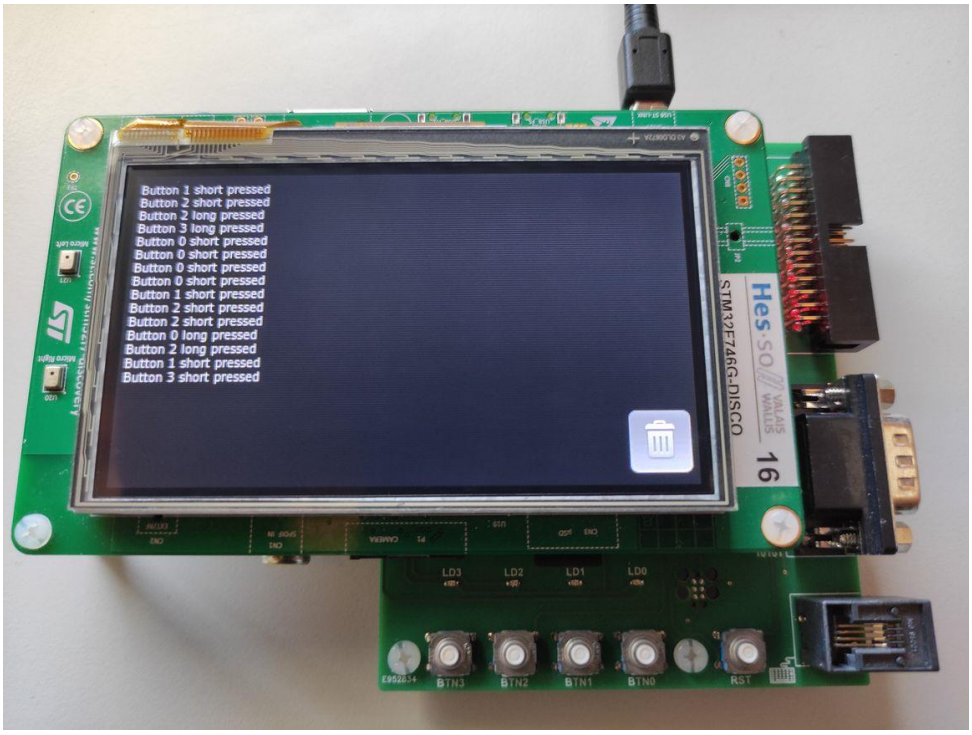
PTR - Labor

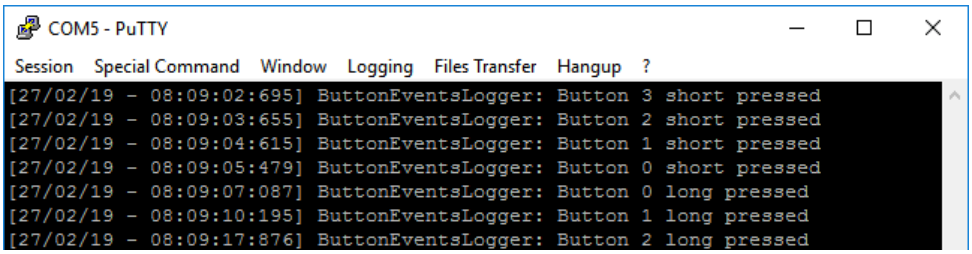
ButtonManager

Introduction

The goal of this laboratory is to get used to some design patterns already seen in the course. With the *ButtonManager* project you will get the opportunity to implement the Singleton, the Subject/Observer pattern and other patterns as well.



Your task is to show on a terminal screen if a button on the extension board of the F7-Disco Embedded System was pressed for a short or for a long time:



HES-SO Valais-Wallis • rte du Rawyl 47 • CP • 1950 Sion 2
+41 27 606 85 11 • info@hevs.ch • www.hevs.ch

Prerequisites

Hardware

For this laboratory you are going to use a F7-Disco Embedded System and a USB cable to connect the system to the development PC.

Development Tools

The STM32CubeIDE software is used to develop, compile, download and debug the code.

To receive and show the messages send over the *virtual com port* (VCP) over USB, you can use PuTTY or any other terminal software able to communicate over a serial port.

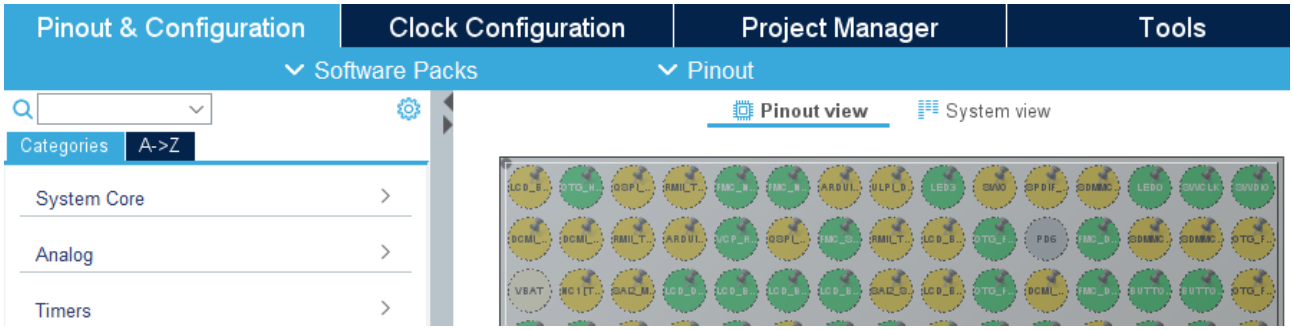
Tool	Comment
STM32CubeIDE	To develop the *ButtonManager* Application for the Embedded System and to configure and generate the code for the MCU peripherals.
PuTTY	To log text output of the Embedded System on the development PC

Software

On the moodle server in the corresponding laboratory section, you will find in the folder *Files/work* a file named **work.zip**. It provides the following information:

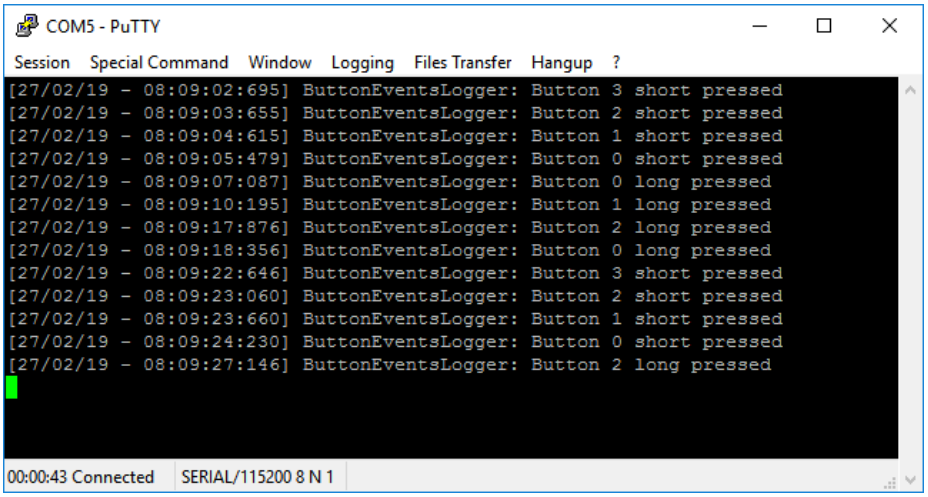
Folder / File	Comment
docs	Folder with documentation
ide-cubeIDE	STM32CubeIDE based project
src/app	ButtonManager application code
src/event	Custom XF events for this project
src/interface	Interface classes provided for this project
src/mdw	Middleware package (with *trace* package)
src/platform	Platform specific code (Embedded System)
src/xf	Execution Framework used in this project (PTR XF)
ide-touchgfx-gen	Source code for the LCD graphical user interface (GUI)

Using the *ButtonManager.ioc* file, located in the STM32CubeIDE project, you can configure and generate the initialization code for the Microcontrollers peripherals:

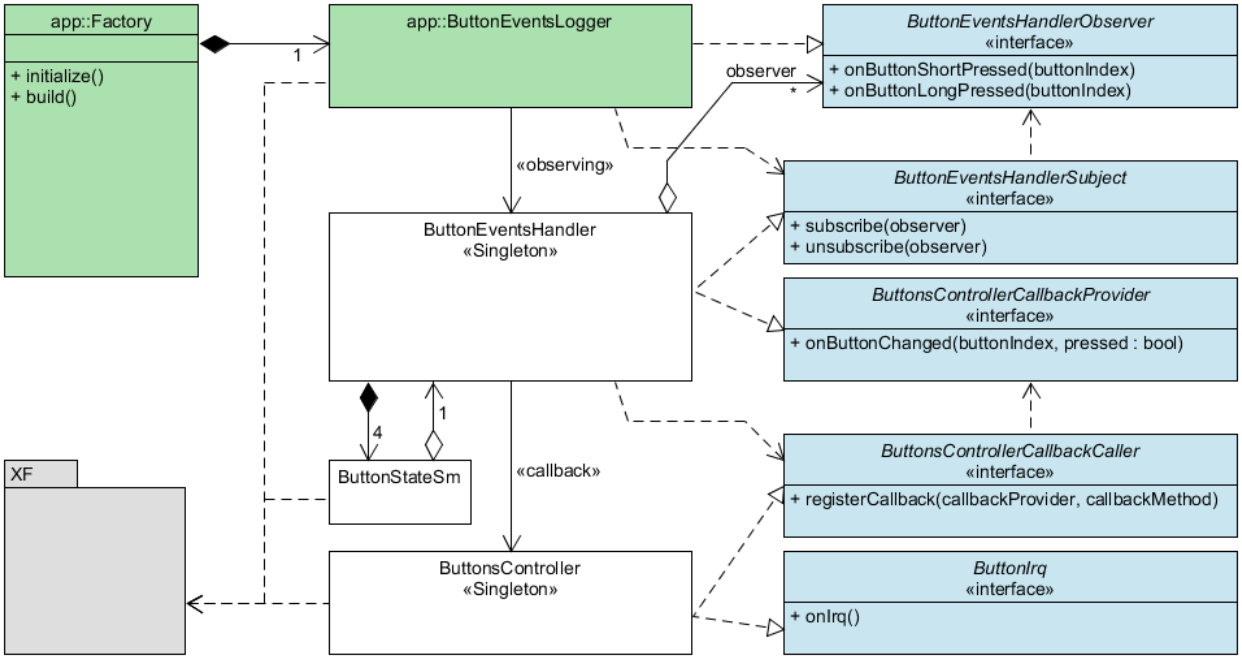


ButtonManager Application

The *ButtonManager* application is responsible to show button short/long pressed events on the host PC via the virtual COM port over USB (Trace functionality):



Following you will find the class model diagram of the ButtonManager application:



Application Constrains

- `ButtonsController` receives button IO interrupts and debounces them
- `ButtonsController` handles all four buttons present on the extension board
- `ButtonsController` **sends button pressed/released notifications** via a callback pattern
- `ButtonEventsHandler` creates button **short pressed and button long pressed notifications**
- `ButtonEventsHandler` notifies via an observer pattern.

- `ButtonEventsLogger` logs the button short/long pressed notifications via Trace to the host PC

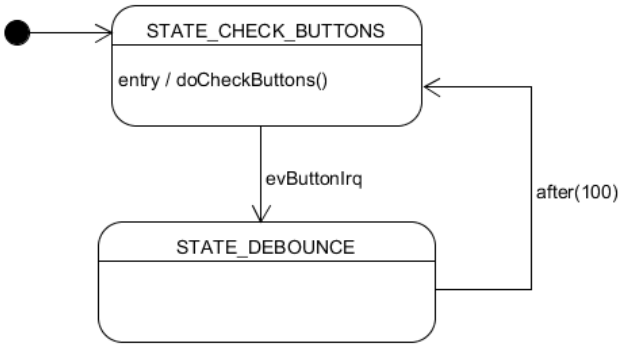
You are going to implement the code for the three classes mentioned above. The interface classes, needed by these classes, are already provided and located in the *src/interface* folder.

Timing Constrains

- Button (GPIO) changes smaller than 100ms are not taken into account (debounced signal)
- A button pressed/released sequence between 100ms and **less than one second** is considered as a "short pressed"
- A button pressed/released sequence taking equal or longer than **one second** is considered as a "long pressed"

State-Machine Diagrams

The following diagrams shows the state-machine of the `ButtonsController`:

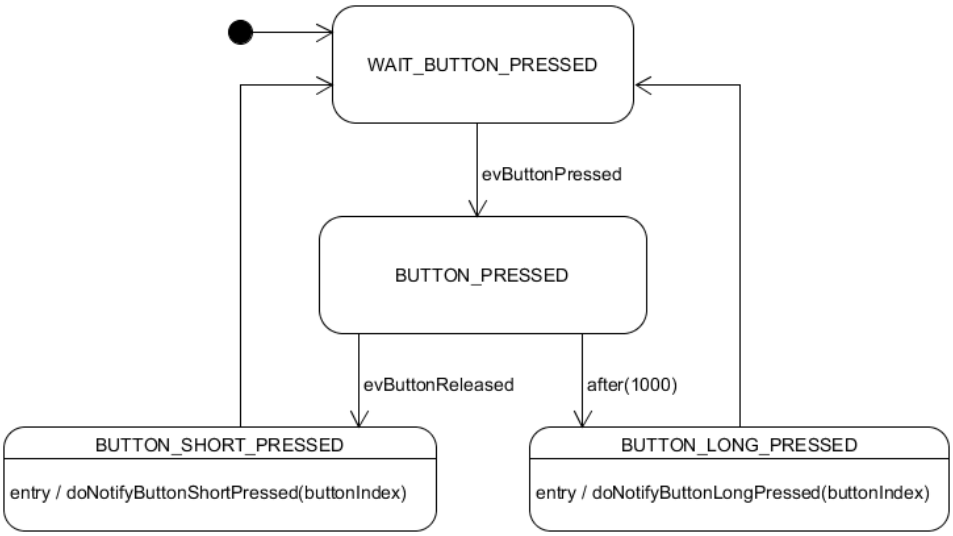


This state-machine is responsible to debounce all 4 GPIO signals at once. The `doCheckButtons()` method evaluates the current GPIO states in comparison to the previously read values and provides the changes to the upper class using the callback method. In our case, it is the `ButtonEventsHandler` class, which gets informed about the button pressed- and released events.

`evButtonIrq` is a static event fired inside the `onIrq()` method. The `onIrq()` method gets called inside the GPIO *interrupt service routine* (ISR) whenever a signal change is detected (*interrupt request* (IRQ) on falling and rising edge).

In this project, the GPIO ISR is located in the 'Core/Src/isrs.cpp' file and named `HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)`.

The state-machine of the `ButtonStateSm` class is used to handle button short-/ long-pressed states of a button:



The methods `doNotifyShortPressed(…)` and `doNotifyLongPressed(…)` provide the short- and long pressed events to the registered observers. The `ButtonEventsLogger` class (and maybe other classes) is receiving these notifications.

ButtonManager Task

Implement the ButtonManager functionality as described above.

Road map

Following you will find a brief listing containing some sub-task. It shall serve as a small guide to achieve the task:

1. Import the *ButtonManager* project into STM32CubeIDE
2. Double-click onto the *ButtonManager.ioc* file to configure the GPIOs for the buttons on the extension board
3. Generate code for the STM32CubeIDE based project
4. Add 'app', 'mdw' and 'platform' packages to the project (virtual folders)
5. Mark the three added packages as source folders so that the containing source files get added to the build process
6. Add `ButtonEventsLogger` class to *app* package
7. Implement `ButtonEventsHandlerObserver` interface on `ButtonEventsLogger`
8. Update the `Factory` class to instantiate a `ButtonEventsLogger` instance
9. Check using a state-machine that the XF is working correctly
10. Check using a state-machine that the Trace functionality is working correctly
11. Add `ButtonsController` class in 'platform/f7-disco-gcc/board/'
12. Implement `ButtonIrq` interface on `ButtonsController`
13. Implement `ButtonsControllerCallbackCaller` interface on `ButtonsController`
14. Implement state-machine for `ButtonsController`
15. Add `ButtonEventsHandler` class in 'mdw/button/'
16. Implement `ButtonsControllerCallbackProvider` interface on `ButtonEventsHandler`
17. Implement `ButtonEventsHandlerSubject` interface on `ButtonEventsHandler`
18. Add `ButtonStateSm` class in 'mdw/button/'
19. Implement state-machine for `ButtonStateSm`
20. Add `ButtonStateSm` instances to `ButtonEventsHandler`
21. Decouple calls to `ButtonEventsLogger::onButtonShortPressed()` and
22. `ButtonEventsLogger::onButtonLongPressed()` by pushing internal events

Nice to Have Tasks

The project manager was so happy about your result that he/she had some great ideas:

- Add `app::ButtonEventsLedFlasher` class which lets the LEDs flash accordingly
  - Button short pressed -> Light up LED for 200ms
  - Button long pressed -> Flash LED twice (2x 200ms)
- Add `app::ButtonEventsFileLogger` class which saves every button short/long pressed into a file located on the SD card

Delivery

Deposit on the Moodle server a zip file containing following information:

- Your ButtonManager project
- A small report showing your realised work (sequence- / activity diagrams, realised sub-tasks, test results, summary, ideas for improvement)