ETR 2021
Lab booklet

Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou

September 19, 2021

Chapter 1

Overview

1.1 Foreword

It is assumed that you have a basic knowledge of the command line. If it is not the case, make sure to have the first configuration steps checked by a member of the teaching team before to proceed with the labs.

> Do not copy the commands from the PDF file, the characters you get, even if the glyph looks the same, may not have the correct code and the shell will not understand them.

When typing shell commands, remember that spaces are important because they separate the command and its arguments. In this document, spaces in commands are represented by a white rectangle like in the following command (it is an example, do not type it):

`cd trampoline` sets the `trampoline` directory as current directory. This assumes the `trampoline` directory is a subdirectory of the current one.

1.2 Setting up the environment

The tools that we will use have already been installed on the machine:

- Trampoline RTOS and the `goil` configuration generator;
- `arm-none-eabi-gcc toolchain` including among other tools a c compiler and a c debugger;
- `stlink`, a set of tools to interact with the STM32F303 microcontroller.

Trampoline is installed in `/opt/trampoline`. `goil` configuration generator is installed in

`/usr/local/bin`. The toolchain and `stlink`, the tool used to download binaries on the board, are installed in `/opt`. All the usefull paths are set in the `.bashrc` startup file of your account.

Now, check that `goil` is working. The command `goil --version` should print:

```
./goil : 3.1.12, build with GALGAS 3.4.3
No warning, no error.
```

Then check that `gcc-arm` is also working. The command `arm-none-eabi-gcc --version` should print:

```
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10.3-2021.07) 10.3.1 20210621 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Last, check that `stlink` tools are available with the following command: `st-info --version`. The output should be:

```
v1.3.0
```

You are now ready to compile and load your first application on the board. But first, let's take a quick look at the board.

Chapter 2

The board

2.1 Overview

The Master CORO lab board is built around a NUCLEO-F303K8 breakout board (BB). A breakout board is a minimal board designed to be used with tiny SMD[1] on a breadboard or in a hobbyist design. The NUCLEO-F303K8 BB is built around a ST Microelectronics microcontroller, the STM32F303K8T6, which has an ARM Cortex-M4 computing core. It is a 32 bits micro-controller running at up to 72MHz. It embeds 64kB of flash memory to store the program and the constant data and 16kB of SRAM to store the variables. Here is the NUCLEO-F303K8 BB:

On the back, a second microcontroller runs the debugging software: ST-LINK. This software establishes a USB communication with the host development computer and allows to download your software to the STM32F303K8T6. It acts also as a JTAG debugger to do step by step execution of the software running on the STM32F303K8T6.

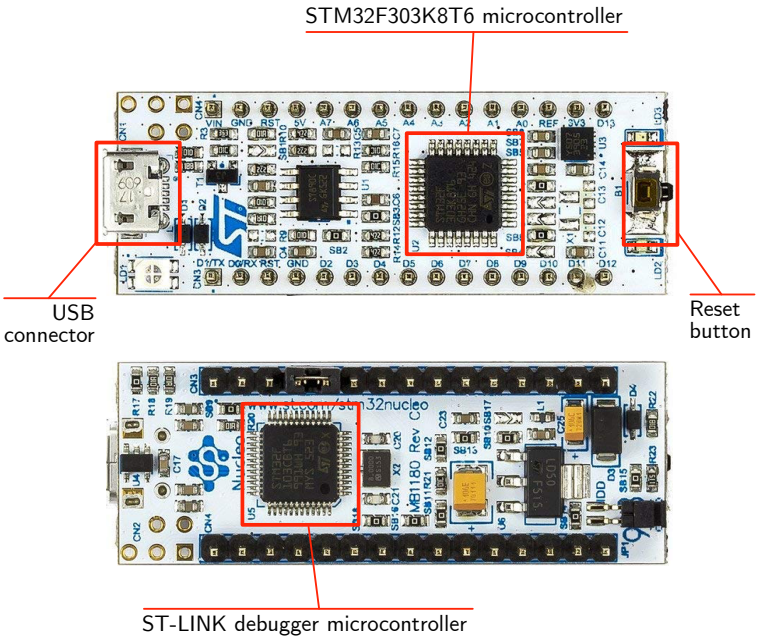
The NUCLEO-F303K8 is pin to pin compatible with the Arduino Nano BB but in the hearth it is very different and of course more powerful.

2.2 The Master CORO board

The board adds the following components:

- A more accessible reset button
- 5 push buttons
- A potentiometer
- A 24 pulses per turn square encoder with a push button
- A 4 positions DIP-switch

[1] Surface Mounted Device



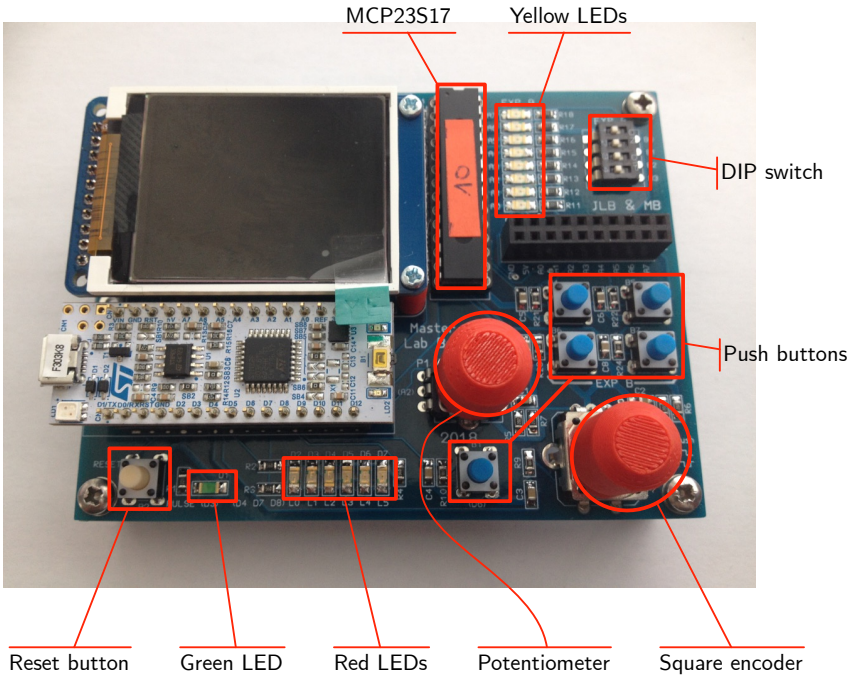
- 1 green LED
- 6 red LEDs connected in charlieplexing
- 8 yellow LEDs
- A graphical 1.8" TFT color screen
- A connector for a servomotor
- A connector for a unipolar stepper motor
- A connector for an ultrasonic range finder
- An external power supply connector

In this list some devices are not connected to the NUCLEO-F303K8 BB but to a Microchip MCP23S17 SPI I/O expander which is an external GPIO where control and data registers are read and updated through the SPI bus: 4 of the 5 push buttons, the DIP-switch and the 8 yellow LEDs.

A picture of the board is presented at figure 2.2

2.2.1 Reset button

The white button in the bottom left corner of the board is the reset button. It is connected to the NUCLEO-F303K8 BB reset, to the MCP23S17 reset and to the 1.8" TFT color screen reset. The reset line is pulled up to the 3.3V supply by a 10kΩ resistor. Pressing the button pulls down the reset line to the ground.



2.2.2 Push buttons

The blue buttons are push buttons. The lone one at the bottom of the board is connected to pin D6 (Arduino) / PB1[2] pin of the NUCLEO-F303K8 BB. The 4 buttons on the right are connected to lines 4 to 7 of port B of the MCP23S17. Il all cases, the pull-up of the GPIO lines should be enabled and pressing the button pulls the line to the ground. Hardware debouncing is wired on all push buttons.

2.2.3 Potentiometer

The central terminal 10kΩ potentiometer is connected to the A2 (Arduino) / PA3 / ADC_IN3 pin of the NUCLEO-F303K8 BB. The two other terminals are connected to the ground and to the 3.3V supply. Turning the potentiometer clockwise decreases the voltage of this pin.

2.2.4 Square encoder

The square encoder has its push button connected to D5 (Arduino) / PB6 which is also the channel 4 of Timer 1. The 2 pins of the encoder are connected to A0 (Arduino) / PA0 / ADC_IN0 and A1 (Arduino) / PA1 / ADC_IN1.

[2] Line 1 of port B.

2.2.5 DIP switch

The DIP switch is connected to lines 0 to 3 of port B of the MCP23S17. The pullup of this lines should be enabled. When a switch is ON, the corresponding line is pulled down to the ground.

2.2.6 Green LED

The green LED named PULSE is connected to pin D3 / PB0 / TIM3_CH3. This LED is on is a high level is set on the pin. Communication between the MCU and both the LCD and the I/O multiplexer uses the SPI interface of the MCU. Thus to ensure integrity of transactions, the software design must enforce that transactions are atomic with regards to each others.

2.3 Yellow LEDs

8 yellow LEDs are connected to PORT A of the I/O expander. They are lit on when the line is set to the high level.

2.3.1 Power supply

The board power is supplied by USB. Connect a USB cable to the MCU USB plug and the board switches on. If a servomotor and/or a stepper motor is connected, the green connector on the back shall be used the poser these devices.

Note about concurrent accesses to the hardware

The SPI bus is shared by the TFT LCD display and the I/O expander. Concurrent accesses to the SPI bus may happen when using the functions to print on the display and/or to read/write ports of the I/O expander. In addition concurrent accesses to the display itself at higher level may happen. If more than one task use the display and/or the I/O expander, it is up to your application to protect both ressources.

2.4 The hardware drivers

A set of functions is provided to use the TFT LCD display, the I/O expander, and one of the embedded timer. The source code of these functions is located within the Trampoline directory in `machines/cortex/armv7em/stm32f303/coroLabBoard`.

Some of them are documented below. For the TFT LCD display, an object, `Tft` is instantiated.

void initCoroBoard() initializes the GPIO and the SPI according to the hardware of the board.

void Tft.erase() erases the TFT display.

void Tft.setTextCursor(*const unsigned int col*, *const unsigned int line*) set the location of the text cursor. *col* argument ranges from 0 to 21 and *line* argument ranges from 0 to 8.

void Tft.print(*any type*) prints at the current cursor location.

void Tft.println(*any type*) same as above and goes at the beginning of the next line.

void Tft.eraseText(*const unsigned int n*) erases n characters from the cursor location.

void setupTimer() sets up timer TIM6 with a 1μs tick. This function has to be called once, before to use the functions related to the timer.

void resetTimer() resets TIM6 value to 0.

uint32_t getTimerValue() returns TIM6 current value.

2.5 The I/O expander driver

The I/O expander drivers implements a set of functions to change the states of LEDs and poll the state of buttons. Notice that the programming of I/O pins in input or output is done in the *setupIOExtender* function.

The driver is provided by the class *mcp23s17*. Users should not create instance of this class as a singleton instance named *ioExt* is created at compile time.

Among the funciton provided by the driver, we will use:

void ioExt.digitalWrite(*port p*, *uint8_t bitNum, bool value*) sets the value of bit *bitNum* (in the range 0 to 7) of port p (either `mcp23s17::PORTA` or `mcp23s17::PORTB`) to value *value* (0 or 1).

uint8_t ioExt.digitalRead(*port p*, *uint8_t bitNum*) returns the value of bit *bitNum* (in the range 0 to 7) of port p (either `mcp23s17::PORTA` or `mcp23s17::PORTB`).

void ioExt.digitalToggle(*port p*, *uint8_t bitNum*) toggles the value of bit *bitNum* (in the range 0 to 7) of port p (either `mcp23s17::PORTA` or `mcp23s17::PORTB`).

void ioExt.setBits(*port p*, *uint8_t bitField*) sets the value of bits of port p contained in *bitField* to HIGH.

void ioExt.clearBits(*port p*, *uint8_t bitField*) sets the value of bits of port p contained in *bitField* to LOW.

void ioExt.writeBits(*port p*, *uint8_t val*) overwrites values of bits of p with *val*.

uint8_t ioExt.readBits(*port p*) returns the values of bits of p.

As an illustration, the following code turn on LED 0 and toggles LED 1.

```
TASK(t_LED0ON_LED1TOGGLE)
{
    ioExt.digitalWrite(mcp23s17::PORTA, 0, 1);
    ioExt.digitalToggle(mcp23s17::PORTA, 1);
    TerminateTask();
}
```

And the following example reads switch button 0 to start Trampoline in a specific appmode.

```
int main()
{
    if (ioExt.digitalRead(mcp23s17::PORTB, 0) == 0)
    {
      /* Start Trampoline in the testAppmode appmode */
      StartOS(testAppmode);
    }
    else                          /* otherwise */
    {
      /* Start Trampoline in the default appmode */
      StartOS(OSDEFAULTAPPMODE);
    }
    return 0;
}
```

Chapter 3

Lab 1 – Periodic tasks

3.1 Goal

The goal of this lab is to become familiar with the development process of application using Trampoline RTOS, and to understand tasks and alarms.

Trampoline includes an OIL compiler. It reads a static description of the objects of the application (tasks, ISRs, events, resources, etc.) and generates the corresponding OS data structures. In addition to the OIL description, the developer must of course provide the C source code of the body of tasxISRs.

3.2 Starting point

Go into the lab1 directory. There are 2 files:

lab1.oil a minimal OIL description.

lab1.cpp a minimal source code.

Edit the lab1.oil file and update the `TRAMPOLINE_BASE_PATH` attribute in function of your configuration: it should point to the directory where Trampoline is installed.

lab1 is a very simple application with only 2 tasks named `task1` and `pulse`. `task1` starts automatically (`AUTOSTART = TRUE { ... }` in the OIL file) and turns on LED A0. `pulse` is activated by an alarm, `alarm_pulse`, every 250 ms. `pulse` toggles the LED named PULSE and display a count on the LCD. Examine the OIL file and the source file.

To compile this application, go into the lab1 directory and type (on a single line):

```
goil --target=cortex/armv7em/stm32f303/coroLabBoard lab1.oil
```

The `--target` option is used to define the target system (here we generate the OS level data structures of Trampoline for a `cortex` core with the `armv7em` instruction set, a `stm32f303` micro-controller and the board is the `coroLabBoard`).

Alongside the C files of the kernel configuration, Goil also generates a build script for the application (files `make.py` and `build.py`).

If you change something in the OIL file or in your source code file, you do not need to re-run Goil because the build script should run it when needed.

Continue the build process by typing:

```
./make.py
```

The application and Trampoline OS are compiled and linked together. The target file is named `lab1.elf`.

To upload the application, check that the board is connected to the computer, then type:

```
./make.py  burn
```

The following message is displayed (some lines have been truncated to fit on the page and the version of st-flash may differ):

```
Nothing to make.
st-flash write lab1.elf.bin 0x8000000
st-flash 1.5.1
2019-04-28T17:45:53 INFO common.c: Loading device parameters....
2019-04-28T17:45:53 INFO common.c: Device connected is: F334 device, id 0x10016438
2019-04-28T17:45:53 INFO common.c: SRAM size: 0x3000 bytes (12 KiB), Flash: 0x10000 ...
2019-04-28T17:45:53 INFO common.c: Attempting to write 16660 (0x4114) bytes to stm32 ...
Flash page at addr: 0x08004000 erased
2019-04-28T17:45:53 INFO common.c: Finished erasing 9 pages of 2048 (0x800) bytes
2019-04-28T17:45:53 INFO common.c: Starting Flash write for VL/F0/F3/F1_XL core id
2019-04-28T17:45:53 INFO flash_loader.c: Successfully loaded flash loader in sram
  9/9 pages written
2019-04-28T17:45:53 INFO common.c: Starting verification of write complete
2019-04-28T17:45:54 INFO common.c: Flash written and verified! jolly good!
```

The program starts (if it do not start press the reset button) and:

- LED A0 should be on;
- LED PULSE should blink;
- A number should increment in the upper left corner of the LCD.

3.3 First add-on

Add a 3rd task identical to the pulse task (except for the name which is pulse2) and an alarm with offset 250 and period 50. This pulse2 task toggles the LED A7.

3.4 Second add-on

Replace the code of the pulse2 task by the code you find in the lab1-add-on directory (this directory is at the same level as lab1).

3.5 Third add-on

Instead of changing the state of LED A3, cancel or start the alarm_pulse. Refer to the OSEK QRDC. You need the GetAlarm, CancelAlarm and SetRelAlarm services.

Chapter 4

Lab 2 – Critical sections : Resources

To show resources usage, we will use a bad program that allows to corrupt a shared global variable which is not protected against concurrent writes.

4.1 Application requirements

The diagram of figure 4.1 describes the application. It is composed of 3 tasks that share 2 global variables **declared with the volatile keyword**: `val` and `activationCount`:

- a background task called `bgTask` which is activated at startup. In an infinite loop it increments then decrements the global variable `val`. This task has priority 1.
- a periodic task called `periodicTask` that activates every 1ms. This periodic task increments the global variable `activationCount`. If `activationCount` is odd, `val` is incremented, otherwise it is decremented. This task has priority 5.
- a periodic task `displayTask` that runs every 1.5 second and prints `val` on the LCD. This task has priority 2.

Run the application and observe the sequence of values of val. Comment.

4.2 Global variable protection

We need to protect access to the shared variable `val`. For that, we use the object `RESOURCE` which works according to the principle of the priority ceiling (Immediate Priority Ceiling Protocol). A resource has a priority that is at least the maximum of the priorities of the tasks that can take this resource. When a task enters the critical section and gets the resource, it has the priority of the resource, and therefore cannot

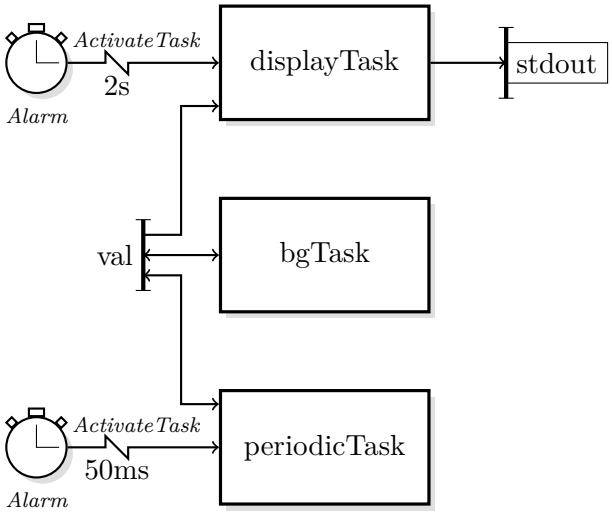


Figure 4.1: Application diagram

be preempted by another task that could access the protected object (it has a higher priority). When the task releases the resource, it resumes its previous priority.

To do this, we declare the object in the file `.oil` (in the section `CPU`, at the same level as the tasks and alarms):

```
RESOURCE resVal {
  RESOURCEPROPERTY = STANDARD;
};
```

In addition, the resource must be declared **in each task that is likely to use this resource**, by adding the line:

```
RESOURCE = resVal;
```

Thus, the compiler *goil* can determine the priority of the resource (the max of the priorities of the tasks that can take the resource).

At the C level, 2 services are available to access/exit the critical section: GetResource and ReleaseResource.

Set up the critical section to access the variable `val`.