PRETS/ETERE
Lab booklet
2019-2020

Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou

March 13, 2020

Chapter 1

Installation

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 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 `.profile` startup file of your account.

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

```
goil : 3.1.9, build with GALGAS 3.3.11
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 Tools for Arm Embedded Processors 8-2018-q4-major) 8.2.1 20181213 [...]
Copyright (C) 2018 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.

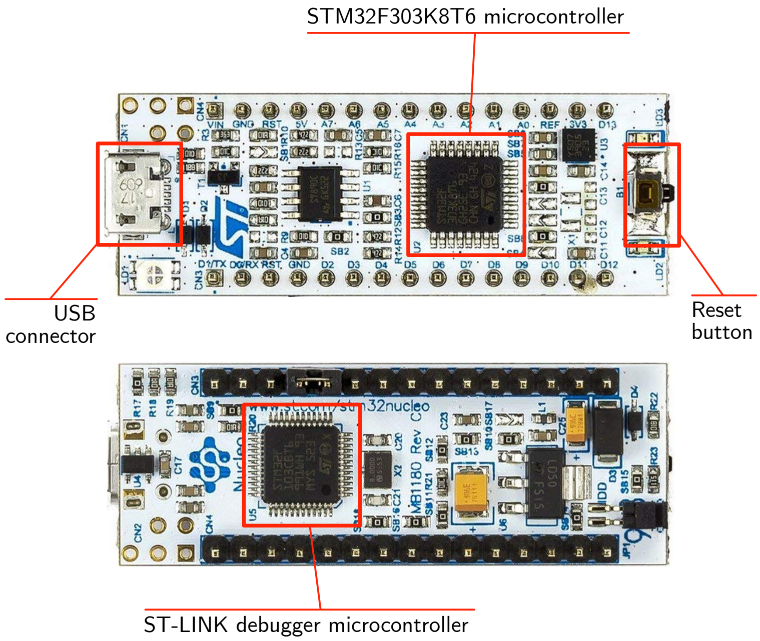
The NUCLEO-F303K8 is pin to pin compatible with the Arduino Nano BB but in the hearth it is very different and of course more powerful.

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

[1]Surface Mounted Device



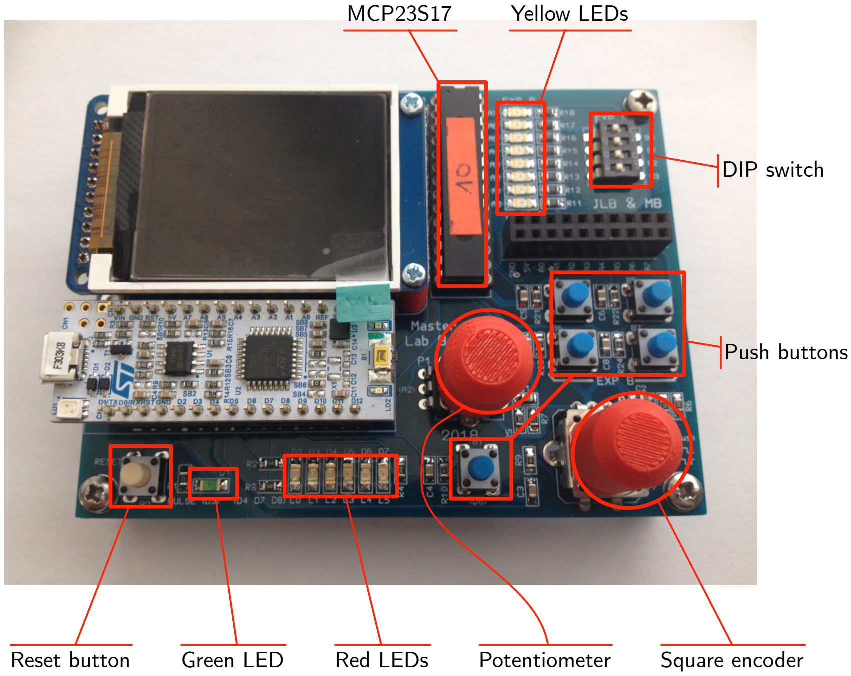
- A 4 positions DIP-switch
- 1 green LED
- 6 red LEDs connected in charlieplexing
- 8 yellow LEDs
- A graphical 1.8” TFT color screen
- A connector for a servomotor
- A connector for a unipolar stepper motor
- A connector for an ultrasonic range finder
- An external power supply connector

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] Line 1 of port B.

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.

2.2.5 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 – Understanding fixed priority scheduling

3.1 Goal

The goal of this lab is to become familiar with the development process of application using Trampoline RTOS, and to understand how fixed priority scheduling works. We will also see Events.

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 tasks and ISRs.

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 1 task named `task1`. It starts automatically (`AUTOSTART = TRUE { ... }` in the OIL file) and turns on LED A0.

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

```
goil --target=cortex/armv7em/stm32f303/coroLabBoard
--templates=/opt/trampoline/goil/templates 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`). The `--templates` option indicates where to find the template files used to generate the configuration of the kernel.

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):

```
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 and LED A0 should be on.

3.3 OS system calls and tasks

For each question, you should create a copy of the lab1 directory and change its name to lab1q1, lab1q2, ...

3.3.1 Task activation and scheduling

The `ActivateTask()` system call allows to activate a task of the application.

Question 1 *Add two tasks in the system:* `task2` *and* `task3`*.*

- *add the declaration of both tasks in the OIL file;* `task2` *should have priority 1 and its AUTOSTART attribute should be set to FALSE;* `task3` *should have priority 8 and its AUTOSTART attribute should be set to FALSE;*
- *in the cpp file, you should:*
  - *declare each tasks with the DeclareTask keyword,*
  - *provide the body of each tasks:*
    - *task* `task2` *turns on LED A2 and turns off LED A3;*
    - *task* `task3` *turns on LED A3 and turns off LED A2;*
  - *and, lastly, modify task* `task1` *so that is activates* `task2` *and* `task3` *(in this order).*

Before to execute the resulting application, draw a schedule and determine the final state of LEDS. Your diagram must clearly show activation, execution and termination of each job using symbols used in the course. Then execute the application and check the correctness of your diagram.

Measuring execution times with a timer

Question 2 *Build an application to measure the execution time of* `ActivateTask` *in the following cases:*

- *fully preemptive scheduling, activation of a higher priority task.*
- *fully preemptive scheduling, activation of a lower priority task.*
- *fully non preemptive scheduling, activation of a higher priority task.*
- *fully non preemptive scheduling, activation of a lower priority task.*

Explain how to use the timer functions for each case and explain the results.

3.3.2 Task chaining

The `ChainTask()` system call allows to chain the execution of a task: the calling job terminates and a new job of the target task is created.

Question 3 *Starting from the application of question 1, replace the call to* `TerminateTask` *by a* `ChainTask(task3)` *at the end of task a task. Update task* `task3` *so that it prints something each time it is executed. Draw a schedule of the new system. Then execute the application to check the correctness of your diagram. Comment on the differences with the previous application.*

Question 4 *Chain to* `task2` *instead of* `task3`*. Update task* `task2` *so that it prints something each time it is executed. Draw a schedule of the new system. Then execute the application to check the correctness of your diagram. Comment on the differences with the previous application.*

Question 5 *Test the error code returned by ChainTask. Modify the OIL file so that ChainTask returns* `E_OK`*. Draw a schedule of the new system. Then execute the application to check the correctness of your diagram. Comment on the differences with the previous application.*

Question 6 *Build an application to measure the execution time of* `ChainTask` *in the following cases:*

- *fully preemptive scheduling, activation of a higher priority task.*
- *fully preemptive scheduling, activation of a lower priority task.*
- *fully non preemptive scheduling, activation of a higher priority task.*
- *fully non preemptive scheduling, activation of a lower priority task.*

Explain how to use the timer functions for each case and explain the results.

3.4 Extended tasks and synchronization using events

For the following exercises, you should use full preemptive scheduling mode.

Unlike a basic task, an extended task may wait for an event. In terms of scheduling, a job is activated when a task is activated or when it leaves the waiting state.

Before to proceed with the following questions, consult the slides of the course to become familiar with events in Trampoline.

Question 7 *Based on the application of Question 1, build an application with the following characteristics:*

- *set priority of* `task2` *to 8 (so priorities are 1 for* `task1` *and 8 for both* `task2` *and* `task3`*);*
- *add two events,* `evt_2` *and* `evt_3`*:*
  - `evt_2` *is set by task* `task1` *to task* `task2`*.*

  - *`evt_3` is set by task `task1` to task `task3`.*

- *modify the code of the tasks:*
  - *task `task1` activates `task2` and `task3` then sets `evt_2` and `evt_3` before to terminate.*
  - *tasks `task2` and `task3` wait for their event, clear it, and terminate.*

Draw a schedule of the new system. Then execute the application to check the correctness of your diagram (before to run the application, add outputs in the code of the task, for instance writes to the LCD or LEDs to ease the correctness checking).

Question 8 *Build an application to measure the execution time of `SetEvent` in the following cases:*

- *fully preemptive scheduling, setting an event to a higher priority tasks that is not waiting for the event.*
- *fully preemptive scheduling, setting an event to a higher priority tasks that is waiting for the event.*
- *fully preemptive scheduling, setting an event to a lower priority tasks that is not waiting for the event.*
- *fully preemptive scheduling, setting an event to a lower priority tasks that is waiting for the event.*

Explain how to use the timer functions for each case and explain the results.

Question 9 *Program an application conforming to the following requirements:*

- *it is composed of two tasks: `server` priority 2, `t1` priority 1.*
- *`server` is an infinite loop that activates `t1` and waits for event `evt_1`.*
- *`t1` prints "I am t1" and sets `evt_1` of `server`.*

Before to run the application, draw a schedule of the execution. Add outputs in the bodies of the task (for instance writes to the LCD or the LEDs) to verify your schedule.

Question 10 *Extend the previous application by adding 2 tasks: `t2` and `t3` (priority 1 for both) and 2 events `evt_2` and `evt_3`. `server` activates `t1`, `t2` and `t3` and waits for one of the events. When one of the events is set, `server` activates the corresponding task again.*

Before to run the application, draw a schedule of the execution. Add outputs in the bodies of the task (for instance writes to the LCD or the LEDs) to verify your schedule.

Chapter 4

Lab 2 – Periodic tasks and Alarms

4.1 Goal

Real-Time systems are reactive systems which have to do processing as a result of events. You have seen in Lab #1 how to start processing as a result of an internal event of the system: by activating a task (`ActivateTask` and `ChainTask` services) or by setting an event (`SetEvent` service). In this lab, you will trigger processing as a result of time elapsing (expiration of an Alarm). This lab uses the following concepts: alarm and counter. On the lab board, the Systick timer is used as interrupt source for alarms. The interrupt is sent every 1ms.

4.2 First application

This application implements a periodic task that reads switch button B0 of the board every 100ms. When switch B0 is open, A3 is turned off. When button B0 is closed, A3 is turned on.

Question 11 *The `updateStateOfB0` function implements a small state machine to convert the state of the switch button to states and events. Draw this state machine. Draw a schedule of the application (the execution of the interrupt service routine associated with the Systick will not be represented; it will be assumed to be negligible). Your diagram should show the instants where the state of the switch changes as well as the state of A3. Build and execute the application to verify your diagram.*

Modify the application: when B0 is closed, a processing is triggered. This processing is performed by activating a job of task `t_process` (priority 3) that, on odd executions, turns on LEDS A4 to A8 and on even executions turns them off.

Question 12 *Draw a schedule of the application. Your diagram should show the physical state of the switch as well as the state of the LEDs (associate numbers with the different state and use these numbers in your diagram). Build and execute the application to verify your diagram.*

4.3 Second application

The second application will use 2 periodic tasks: `t1` (priority 2, period 1s) and `t2` (priority 1, period 1.5s). t1 toggles LED A3 each time it executes and t2 toggles LED A4.

Question 13 *Draw a schedule of the application that show the 20 first states of the LEDs. What is the global period of the system?*

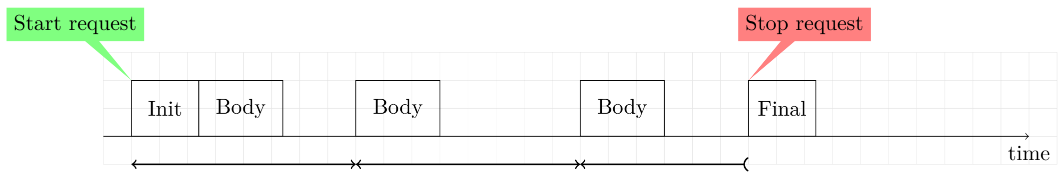
The application needs a counter and 2 alarms. Program, build, and run the application.

4.4 Third application

In the third application, alarms, counters, and polling of the switch buttons are mixed. This application is a system with 2 switches. After startup, the system is waiting. When the first switch is closed, the system starts a *function*[1] F that is implemented using a periodic task (period = 1s). To "see" F, uses a blinking LED. When the first switch is opened, function F is stopped. When the second switch is closed, the system is shutdown as quickly as possible (ie ShutdownOS[2] is called).

Question 14 *Explain the design of the application: provide the list and configuration of tasks and alarms. Draw schedules showing different executions. Build and execute the application to verify your diagrams.*

Requirements change. Now function F implementation needs an Init code (runs once when F is started) and a Final code (runs once when F is stopped) as shown in the following diagram:



[1] Here we mean a function of the application, not a function of the C language.

[2] The ShutdownOS service shutdowns the operating system. All tasks and alarms are stopped. ShutdownOS takes a StatusType argument to specify the kind or error which occurred. In our case use E_OK as argument.

Question 15 *Modify the application to take the new requirements into account. Your implementation will use only basic tasks. Each phase of F (Init, Body, and Final) is executed in a different task. Explain the design of the application: provide the list and configuration of tasks and alarms. Draw schedules showing different execution scenarii. Build and execute the application to verify your diagrams.*

Question 16 *Same question as above, but your implementation will use an extended task to control the execution of Init, F, and Stop.*

4.5 Fourth application

In this part, you will implement a watchdog. It is a mechanism that allows to stop a processing or a waiting period once a delay has elapsed.

In your application, each time B0 is closed, B3 must be pressed within 4 seconds. In this case, you print the time between the two occurrences. Otherwise, an error message is displayed. If B0 is closed more than once are within 5 seconds from the first time, these events are ignored.

Question 17 *Specify this behaviour with a state machine. What is happening if the timeout occurs just after B3 has been closed but before the waiting task got the event? Design and program a solution that handles this situation correctly. Draw a schedule of the behaviour of your application in such a scenario (this schedule should show the state of the alarms).*

4.6 Fifth application

Program a chase[3] with a 0.5s period. To do so, use 8 periodic tasks. Each periodic task manages a LED. The chase effect is done by using alarms with a time shift between them.

When B0 is closed, the chase starts, when it is opened, it is reset. When B3 is closed, the chase stops until B3 is opened. When B2 is closed, the chase direction changes (even if it is stopped).

Question 18 *Specify the high level behaviour with a state machine. Design and program the application. Explain your design.*

[3] chenillard in French

Chapter 5

Lab 3 – Shared object access protection

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. This has been presented in the course. This lab will show different ways to prevent this wrong behavior by using resources (standard and internal) or other solutions (preemption and priority).

To ensure that the compiler does not hide our bug, update the value of the `CFLAGS` key in the OIL file: replace option `-Os` by `-O0` to turn off all optimizations.

5.1 Application requirements

The diagram of figure 5.1 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`, 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 a job every 50ms. This periodic task increments the global variable `activationCount`. If `activationCount` is odd, `val` is incremented, otherwise it is decremented. This task has priority 10.
- a periodic task `displayTask` that runs every 2 second and prints `val` on the LCD. This task has priority 2.

Question 19 *Before programming the application, gives the expected sequence of values for val. Design, program, and run the application. Now, decrease gradually the period of task `periodicTask` and observe the sequence of values of val. Comment.*

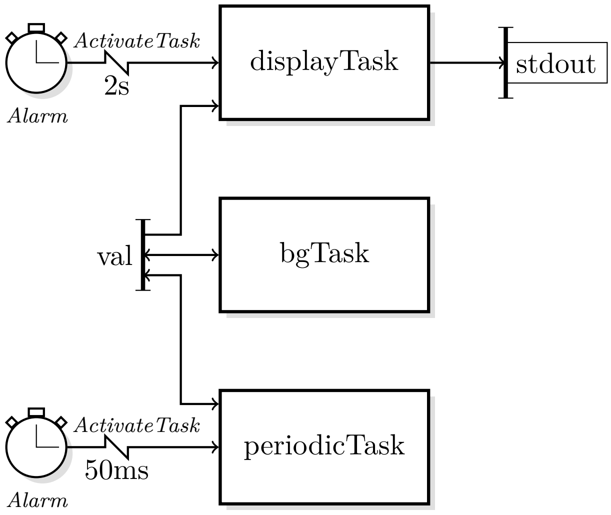


Figure 5.1: Application diagram

5.2 Global variable protection

Update the OIL file and the C program to protect the access to the global variable `val`. Use a resource to do it.

The resource priority is automatically computed by goil according to the priorities of the tasks which use it.

The OIL compiler generates many files in the directory bearing the same name as the oil file (without the .oil suffix). Among them 3 are of interest for this lab:

- `tpl_app_define.h`
- `tpl_app_config.h`
- `tpl_app_config.c`

The file `tpl_app_config.c` contains the tasks descriptors among other data structures. These structures are commented.

Question 20

- *Recall the computation rule of the ceiling priority of a resource with the immediate ceiling protocol. According to this rule, what should be the ceiling priority of the resource?*
- *Find the actual ceiling priority in `tpl_app_config.c`. Is it the expected value? If not, is it a problem?*

5.3 Protection with an internal resource

An internal resource is automatically taken when the job gets the CPU, and released when it terminates. Replace the standard resource by an internal resource in the OIL file. Remove the `GetResource` and `ReleaseResource` in the C file.

Question 21

- *What happens? Why?*
- *How to solve the problem? Draw a schedule of a correct implementation of the system. Program this solution.*

5.4 Protection using a single priority level

Question 22 *Modify the OIL file: remove the resource and set the priorities so that all tasks share the same priority. Draw a schedule of the system.*

5.5 Protection using fully non preemptive scheduling mode

Question 23 *Modify the OIL file: all tasks are now non preemptive. Draw a schedule of the system.*

5.6 Upgrading the coro_utils function

Question 24 *As explained in section 2.4, both the screen and the IO expander use the SPI interface of the MCU. Thus, to ensure coherency of the transactions, we have avoided using both devices in the same application. Upgrade the current implementations of the functions to solve this problem and design an application that shows the effectiveness of your solution.*

Chapter 6

Lab4
Internal communication

6.1 Overview

Internal messaging may be used as an easy to use replacement for shared variables. Messaging allows to send data between tasks. This lab will show the different ways to communicate in OSEK.

Design a first simple application with 2 communicating tasks. The OIL file declare 2 tasks: `sender` and `receiver` and 2 messages: `outMessage` and `inMessage`. `inMessage` is connected to `outMessage`. Task `sender` uses `outMessage` and task `receiver` uses `inMessage`.

Task `sender` is activated every second by the mean of an alarm and sends data to `outMessage`. Send for instance the number of activation of task `sender`. When the data is received in `inMessage`, the notification mechanism activates task `receiver`. `receiver` reads the data from `inMessage` and prints the value.

Notice that message identifiers must be declared before to be used in the source file with `DeclareMessage`.

6.2 Message broadcasting

OSEK messaging supports *many-to-many* communication. This is done by having more than one receiving messages connected to the same sending message.

Question 25 *Add 2 other receiving messages connected to `outMessage`. The first message activates the task `Emergency` (add the corresponding task), the second message sets* *an event to task `NormalOperation` (add the corresponding extended tasks too). Write the C code of the tasks you added. The first one receives the message and prints the value; the second one waits for the event in an infinite loop, receives the message and prints the value. Check the application works as expected.*

6.3 Filtering

Question 26 *Add a filter to activate task `receiver` upon the 4th message and then every 3 values (ie 4th, 7th, 11st and so on).*

Question 27 *Modify task `sender` to send pseudo-random values ranging from 0 to 100 (you can use for instance xorshift32 as described here: https://en.wikipedia.org/wiki/Xorshift) and then a modulus. Using filtering, set the event to task `NormalOperation` only if the value is within the range* $[20, 60]$*; activate task `Emergency` only if the value is not within the range.*

6.4 Queued messages

Question 28 *Restarting with the Messaging application, modify the `inMessage` to make it a queued message with a size of 4 data items and remove the notification. Add an alarm to activate task receiver every 4 seconds. Check the return value of `ReceiveMessage` and verify that no error occurred (queue under- or over-flow). If errors occur, correct the application.*

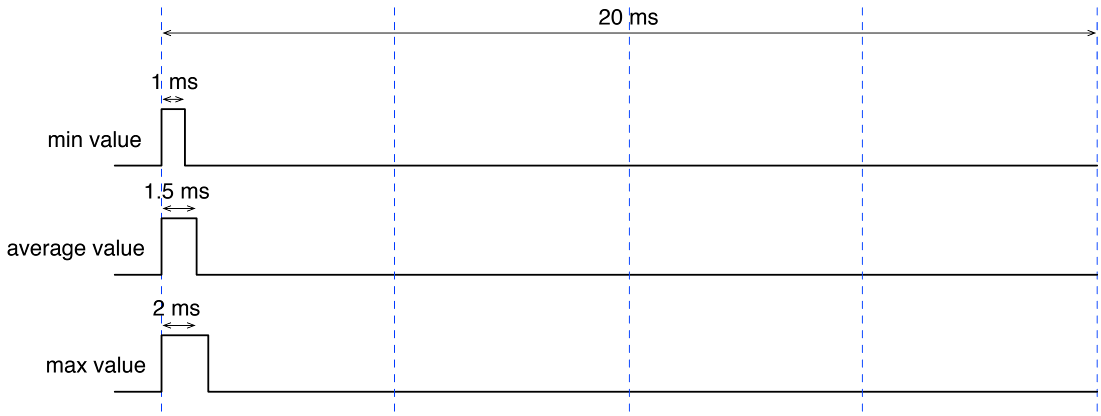
Chapter 7

Lab5
Driving servos

The goal of this lab is to build a small application to drive a servomotor. Get the `servo_start` package from https://github.com/TrampolineRTOS/Labs.

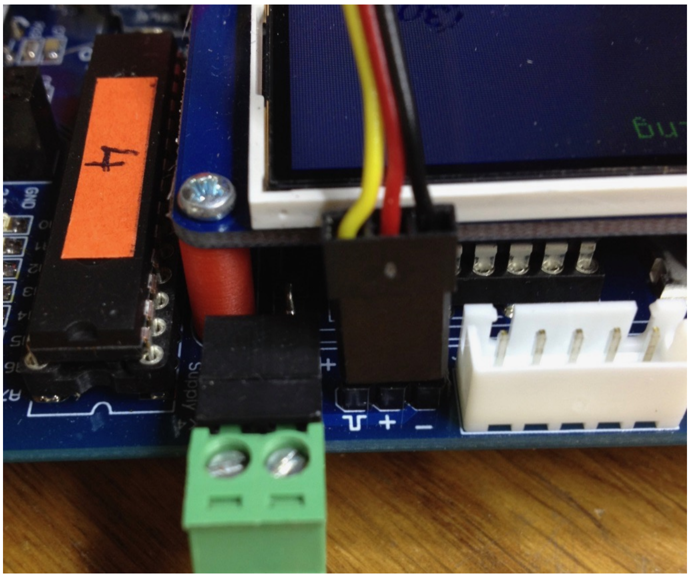
7.1 How servos work

Servos are driven by using a PWM signal. The width of the signal at the high logic state specifies the position of the servo. The width may range from 0.5ms to 2.5ms for a 180° rotation. In practice servos may not be able to turn by 180° and we are going to use a pulse width ranging from 1ms to 2ms. The signal should repeat every 20ms.



7.2 Servo PWM generation

The board has a 3 pins connectors for servos on the back side between the external power supply connector and the pair of white JST connectors. There is one symbol per pin: $-$, $+$ and ⎍. The Servos must be connected with the **yellow wire** on the ⎍ pin and the **black wire** on the $-$ pin. as shown on the picture below.



The 3.3V regulator of the Nucleo BB is not able to power the servo. We need an external power supply connected to the black power supply connector on the back of the board. Get a power supply cord and connect it to the power supply (9V) and to the board.

We are going to drive 1 servo with a task named `t_servo`.

Question 29 *Program task `t_servo`. `t_servo` is periodic with a 20ms period. `t_servo` toggles the green LED, D1 (D3) (PORTB 0[1]). Verify the period using the scope[2].*

Now instead of toggling the LED, `t_servo` will use the function `setServoPulse`. This function takes 1 argument, the `duration` of the pulse in μs. `setServoPulse` does not allow values lower than 1000 and greater than 2000 in order to avoid to damage the servo. `setServoPulse` works as follow: the `ouput` is set to 1 and a timer of the microcontroller is programmed to generate an interrupt after `duration` μs. In your OIL file, an ISR which is located in the `servo.c` file is defined as follow:

```
ISR isr_timer_7 {
  CATEGORY = 1;
  PRIORITY = 10;
  SOURCE = TIM7_IRQ;
```

[1] Use digitalToggle(PORTB, 0).

[2] Plug a wire from the ground of the probe into the leftmost black connector terminal (labelled GND) and touch the LED left terminal with the probe.

```
};
```

Question 30 *Modify `t_servo` to use `setServoPulse` and generate a pulse with a width equal to 1.5ms. Verify the behavior using the scope*[3].

Then connect the servo, it should go to the average position.

7.3 Adding high level behavior

Now we want to set the position of the servo. To do that we need 1 more task that sets the position in a global variable. The corresponding variable is read by the `t_servo` task to drive the servo.

Question 31 *First we want the position of the servo incremented until it reaches its maximum position (2ms pulse), then decremented until it reaches its minimum position (1ms pulse). So the servo does round trips. The time we want to go from the minimum position to the maximum position is 5s. Assuming the position is incremented and decremented by one, what is the period of theses tasks in counter tick unit ?*

Now we want to be able to set the minimum and maximum positions of the servo. To do that we use buttons.

Question 32 *Add to the previous application the following features and draw an automaton to model the application:*

When button B4 is pressed, the servo stops its round-trip. If it is pressed again, round-trip continue. When the round-trip is stopped the following functions are available:

When DIP switch B0 is ON, the minimum position is selected, the servo goes to this position. Then push buttons B7 decrements the minimum position and B5 increments the minimum position.

When DIP switch B0 is OFF, the maximum position is selected, the servo goes to this position. Then push buttons B7 decrements the maximum position and B5 increments the maximum position.

Minimum position should be kept greater or equal than the position corresponding to the 1ms pulse. Maximum position should be kept lower or equal than the position corresponding to the 2ms pulse. Minimum position should be kept lower or equal than the maximum position.

When button B4 is pressed again, the servo continues its round-trip and respects the minimum and maximum positions that have been set.

[3] Plug a wire from the ground of the probe into the leftmost black connector terminal (labelled GND) and touch ⎍ pin with the probe.