PRETS/ETERE
Lab booklet
2019-2020

Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou

May 11, 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 virtual machine:

- Trampoline RTOS and the `goil` configuration generator;
- GNU development toolchain including a C compiler and linker (`gcc`), a debugger (`gdb`) and an implementation of standard C library (`glibc`).

Trampoline is installed in `/opt/trampoline`. `goil` configuration generator is installed in `/usr/local/bin`. 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 : 3.1.11, build with GALGAS 3.3.12
No warning, no error.
```

Chapter 2

The virtual platform

2.1 Overview

An embedded real-time system interacts with its environment (the plant that it drives) through sensors and actuators. For this online version of the lab, we will not have access to a real microcontroller-based platform equiped with such devices. Instead, we are going to use a virtual platform that allows to run TrampolineRTOS application within a Unix process and mimic the behaviour of a very simple environment through a terminal. This virtual platform is automatically embedded in the program generated by TrampolineRTOS build process when the option `--target` is given to `goil`.

The virtual platform is equiped with:

- A set of **timers** that can be connected to alarms through counters to achieve periodic behaviours or watchdogs. The tick of these timers is set to 10 ms.
- Three **buttons** simulated by keyboard keys:
  - `'a'`: The virtual platform raises an interrupt each time it is pressed through Unix signal `SIGTERM`.
  - `'b'`: The virtual platform raises an interrupt each time it is pressed through Unix signal `SIGTRAP`.
  - `'q'`: The virtual platform and the TrampolineRTOS application are shut down each time it is pressed.
- Four **leds**: `RED`, `GREEN`, `BLUE` and `YELLOW`. The state of the leds can be changed through the following functions:
  - `void set_leds(uint8_t leds)`: switches on the leds given as arguments. For instance, to switch on leds `GREEN` and `BLUE`, you can use `set_leds(GREEN | BLUE)`.

  - `void reset_leds(uint8_t leds)`: switches off the leds given as arguments. For instance, to switch off leds `RED` and `YELLOW`, you can use `set_leds(RED | YELLOW)`.

  Each time one of these function is called, the state of the leds is printed on the terminal.

- A **display**: the control terminal of the process running TrampolineRTOS program and that can be used through functions of the standard C library.

2.2 Important remarks

The virtual platform has been packaged in a hurry to create an online version of the labs that can be run during the COVID-19 crisis. It cannot be considered as a robust and well-tested piece of software. In order to use it without breaking everything please take into account the following remarks:

- Although the scheduling of tasks and ISR within the TrampolineRTOS application conforms to OSEK/AUTOSAR OS scheduling policy, remember that this application is embedded in a Unix process that is itself scheduled alongside a bunch of other processes by the underlying non real-time OS. Thus, the application are not run in real-tile wrt. physical (wall-clock) time.
- Moreover, timers of the virtual platform suffer from important jitters due to the fact that (i) the underlying host operating system is not a RTOS, and (ii) Unix signal delivery semantics is piggy-backed on other OS activities and do not trigger preemption by themselves. Thus, the application can be notified of the expiry of a timer with a significant tardiness.
- Do not use input functions from the standard C library as it could interact badly with the virtual buttons. Generally speaking, try to avoid using the C standard library that is most of the time not (fully) available on an embedded target.
- To properly end a line of text to be displayed, you have to use both carriage return and line feed (ie. `\r\n`).

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 prints ”`Hello world`” on the terminal.

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

```
goil --target=posix --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 the `posix` target). 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_exe`.

To run the application, simply run the application:

```
./lab1_exe
```

The program starts and prints the message. As soon as the job of Task1 is terminated, Trampoline gets into idle (starting an internal idle task). To terminate the process, simply type `q`.

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 description file (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 implementation file (C file), you should:*
  - *provide the body of each tasks:*
    - *task `task2` prints "==Task2==" on the terminal;*
    - *task `task3` prints "==Task3==" on the terminal;*

  - *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 give the text written on the terminal 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.

Trace evaluation

It is not possible to measure timings on this virtual platform. However, The trace allows an analysis of the behavior after execution (post-mortem). To get the trace (after an execution) on standard output:

```
./readTrace.py
```

To redirect the output to a file (`output.txt` here), simply write the command:

```
./readTrace.py >output.txt
```

There is one event per line, beginning with the current date (based on `SystemCounter` value).

Even if the date is the same for several lines, the events (1 per line) are recorded chronologically in the trace file.

Analyze the trace of the current application run, and report on the gantt diagram each events on tasks states.

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 2 *Starting from the application of question 1, replace the call to* **`ActivateTask(task3)`** *and* **`TerminateTask`** *by a* **`ChainTask(task3)`** *at the end of task* **`task1`**. *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. Explain the differences with the 2 traces.*

Question 3 *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 4 *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 5 *If we wanted to measure the execution time, of `ChainTask`, we would have to use a timer as stopwatch. Explain where it should be started and stopped 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 6 *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, print their name on stdout (`==Taskx==`) and terminate.*

Draw a schedule of the new system. Then execute the application to check the correctness of your diagram and check that the output order the one awaited. Then, in a second step, explain the trace generated and report each trace event to the schedule of the tasks.

Question 7 *As in the `ChainTask()` service call, give the location where to start and stop a stopwatch to measure the `SetEvent` service call duration 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.*

Question 8 *Program an application conforming to the following requirements:*

- *it is composed of two tasks: `server` priority 2, `t1` priority 1.*
- *`server` is `AUTOSTART` and built around 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 to verify your schedule.

Question 9 *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 to verify your schedule.

Chapter 4

Lab 2 – Periodic tasks, Alarms, and ISRs

4.1 Goal

Real-Time systems are reactive systems which have to trigger processing at certain dates or when the controlled plant reaches a certain state. 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 or notification of an external event. This lab uses the following concepts: counters, alarms, interrupt request, and interrupt service routine.

4.2 First application

You start with a working application composed of an alarm, a task and an isr.

Let us start by analyzing the OIL configuration file. The alarm `oneSec` is `AUTOSTART`, so it is activated at startup. Its `CYCLETIME` and `ALARMTIME` are set to 100 counter ticks, so it will take its action every 1 second. The action is `ACTIVATETASK` and targets task `task1`, so `task1` will be activated every 1 second. There is also an interrupt service routine (isr for short) named `when_a` that is executed whenever an interrupt request is raised on source `SIGTERM`, ie. when key `a` is pressed in the virtual platform.

Let us now look at the C code. Whenever isr `when_a` is executed, it toggles the state of LED `GREEN`. Whenever task `task1` is executed (every 1 second), it toggles the state of LED `RED`.

Question 10 *Update the application so that, while still updating the state of LEDs* `GREEN` *and* `RED` *as above, it also toggles the state of LED* `BLUE` *whenever key* `b` *is pressed in the virtual platform.*

4.3 Second application

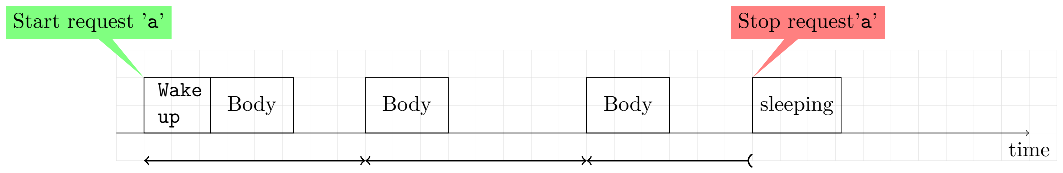
The second application will use 2 periodic tasks: `task1` (priority 2, period 1 second) and `task2` (priority 1, period 1.5 second). At each activation, `task1` toggles LED `GREEN` while `task2` toggles LED `RED`.

Question 11 *Draw a schedule of the application between date* 0 *and* 12 seconds *that also shows the state of the LEDs. Design, program, and run the application.*

4.4 Third application

In the third application, a *function*[1] is controlled by external events. The external event is key `a` has been pressed. When it occurs, it switches the state of the function from active to inactive and conversely. When the functions becomes active, it must execute a wake up phase. To visualize this step, print message `wake up` on the terminal. Then, the function performs a periodic processing. To visualize this step, blink a led every 500 ms. Lastly, when it becomes inactive, it must execute a sleeping phase To visualize this step, print message `sleeping` on the terminal.

Another external event is used to shutdown the system as fast as possible. This event is key `b` has been pressed. The shutdown sequence should switch the function to inactive if it is not already in this state before to shutdown the system (using the `ShutdownOS` service).



Question 12 *Propose a design that uses only basic tasks. Draw schedules showing different executions of your design. Program, build and test your design.*

Question 13 *Propose a design that uses events and extended tasks. Draw schedules showing different executions of your design. Program, build and test your design.*

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

4.5 Fourth application

In this application, you will implement a watchdog. A watchdog is a mechanism used to take an action after a certain time has elapsed, either to interrupt some on-going processing, or to bound a waiting period.

In your application, each time key `a` is pressed, then key `b` must be pressed within 4 second. In this case, you print the time between the two events. Otherwise, an error message is displayed. When key `a` is pressed, subsequent presses on key `a` are ignored for the next 5 second.

Question 14 *Specify this behaviour with a state machine. What is happening if key* `b` *is pressed precisely* 4 second *after key* `a`*? Design 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). Program and test your design.*

4.6 Fifth application

In this application, you will program a chase between the LEDs. In a chase, at any time, only one LED is on and the chase propagates periodically in a given direction. Your chase will propagate with a period of 500 ms, with left to right as its initial direction.

It will be controlled through keys:

- When key `a` is pressed, it re-initializes the chase. If it was not active then it becomes active.
- When key `b` is pressed, if the chase is active, it is paused. If it was paused, then it is resumed but in the opposite direction.

Question 15 *Describe the system with a state machine. Design a system that implements this state machine. Program and test your design.*

Chapter 5

Lab 3 – Shared object access protection

To show how to propely access to shared resources, 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,`val` and `activationCount`, **declared with the volatile keyword**. The tasks in the system are:

- 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`. This task has priority 2.

Question 16 *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.*

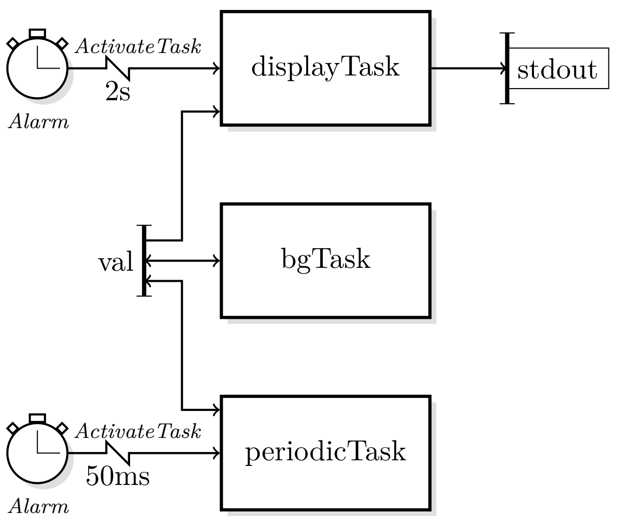


Figure 5.1: Application diagram

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 task descriptors among other data structures. These structures are commented.

Question 17

- *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 18

- *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 19 *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 20 *Modify the OIL file: all tasks are now non preemptive. Draw a schedule of the system.*

Chapter 6

Lab4
Internal communication

6.1 Basic usage

Internal communication allows task to exchange data under the control of the RTOS. It can be used in the place of communication through shared global variables. It is usually preferable to use internal communication rather than shared global variables because (i) it tends to improve application design by making data flows explicits, and (ii) it limits the bugs resulting by a wrong use of synchronization tools.

In the starting application for this lab, task `send` sends a message to task `receive`. The former is activated at system startup, while the latter is activated when a message is delivered. Build, run, and study the code of this application to gain basic knowledge of internal communication. More details are provided in the course and in the reference documents available on TrampolineRTOS github page (bottom of homepage).

Question 21 *Design an application with two periodic tasks (period* 1 second *and* 1.5 seconds*). At every activation, each task sends a message containing its number of activation since system startup. A third task is activated whenever one of this messages arrives and prints the identity of the sender and its number of activation. The receiver task shall process only one message per activation. It shall not miss any message.*

Explain your design and draw a schedule. Program and test your design.

6.2 Many-to-many communication pattern

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

Question 22 *Design an application with a periodic task that sends random data with values in* $[0, 100]$*. Two other tasks are receiving this flow:*

- *The first one is activated whenever a new message is available and print its value;*
- *The second one is an extended task that is notified of the arrival of a new value by an event. When the event is received, it prints the average value over the last 10 messages (or less if less than 10 messages have been received so far).*

Explain your design and draw a schedule. Program and test your design.

6.3 Filtering

OSEK messaging also supports filtering. Filters are predicate attached to sending or receiving message objects, used to decide if data should be sent or received.

Question 23 *Add the following filters to your application:*

- *The task activated upon message arrival is now acticated only if the value is below* 20 *or above* 80*. In the former case it prints message* `low val` *and in the latter one it prints* `high val`*.*
- *The other task is notified only one every two messages. Notice that it should nonetheless process all messages.*

6.4 Application design

Question 24 *Propose a new design for the application described in section 5.1 that uses messages instead of shared variables.*

Explain your design and draw a schedule. Program and test your design.