Real-time programming labs
IMTA, option AII, 2018-2019

Jean-Luc Béchennec, Sébastien Faucou

November 18, 2018

Chapter 1

Installation

1.1 Foreword

It is assumed that you have a basic knowledge of the Linux environment. If it is not the case, make sure to have the first configuration steps checked 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.

A toolchain to build and download programs with Trampoline has been installed on the computers in the `/usr/local/tp_treel_imta`. In this folder, you will find:

- a version of Trampoline in the `trampoline` subfolder; it includes the `goil` compiler to parse `OIL` kernel configuration files, the kernel code, and an automated build system.
- a cross-compilation chain, namely `arm-gcc` in the `gcc-arm-noe-eabi-7-2018-q2-update` subfolder.
- a tool to download binaries to the target board: `teensy-loader-cli`.

All these tools are open-source (see the licence of each projet for more details) and are available on the following websites:

- https://github.com/TrampolineRTOS/trampoline
- https://launchpad.net/gcc-arm-embedded
- https://github.com/PaulStoffregen/teensy_loader_cli

1.2 Environment settings

In order to be able to use the tools, you have to define the configuration of your development environment. First, you have to inform the shell of the location of the tools. This is done by setting the `PATH` environment variable.

Edit the `.login` file in your home directory add the following lines at the end:

```
set path=($path /usr/local/tp_treel_imta/bin /usr/local/tp_treel_imta/gcc-arm/gcc-arm/bin)
```

The `.login` file is parsed by the shell when it starts. To force your current shell instance to read the file, use the following command:

```
source ~/.login
```

This will enable the configuration in the terminal where you typed the command. In order to have the configuration active in all terminals, you have to log out and then log in the system.

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

```
goil : 2.1.26, build with GALGAS 3.1.3
```

Test that gcc-arm is also working. The commande `arm-none-eabi-gcc --version` should print:

```
arm-none-eabi-gcc (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 7.3.1 20180622 (release) [ARM/embedded-7-branch revision 261907]
Copyright (C) 2017 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, test Teensy loader is working. The command `teensy-loader-cli` should print:

```
Filename must be specified

Usage: teensy_loader_cli --mcu=<MCU> [-w] [-h] [-n] [-b] [-v] <file.hex>
        -w : Wait for device to appear
        -r : Use hard reboot if device not online
        -s : Use soft reboot if device not online (Teensy3.x only)
        -n : No reboot after programming
        -b : Boot only, do not program
        -v : Verbose output

Use ‘teensy_loader_cli --list-mcus‘ to list supported MCUs.

For more information, please visit:
http://www.pjrc.com/teensy/loader_cli.html
```

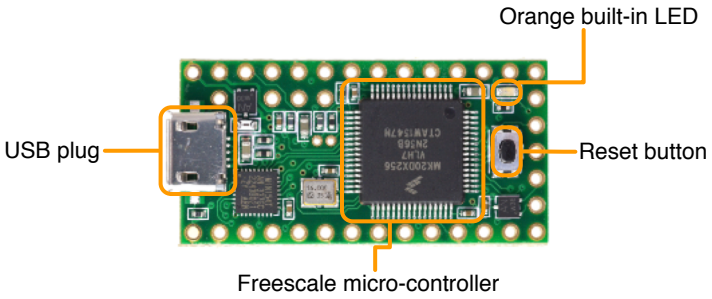
You are now ready to compile and load your first application on the board. But first, let's take a look at the board.

Chapter 2

The board

2.1 The Teensy 3.1

The board is built around a Teensy 3.1 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 Teensy 3.1 BB is built around a Freescale Micro-controller, the MK20DX256VLH7, which has an ARM Cortex-M4 computing core. It is a 32 bits micro-controller running at 96MHz. It embeds 256kB of flash memory to store the program and the constant data and 64kB of SRAM to store the variables. The Teensy is built by PJRC, a small company from the USA. The web site is here. Here is the Teensy 3.1:



2.2 The labs board

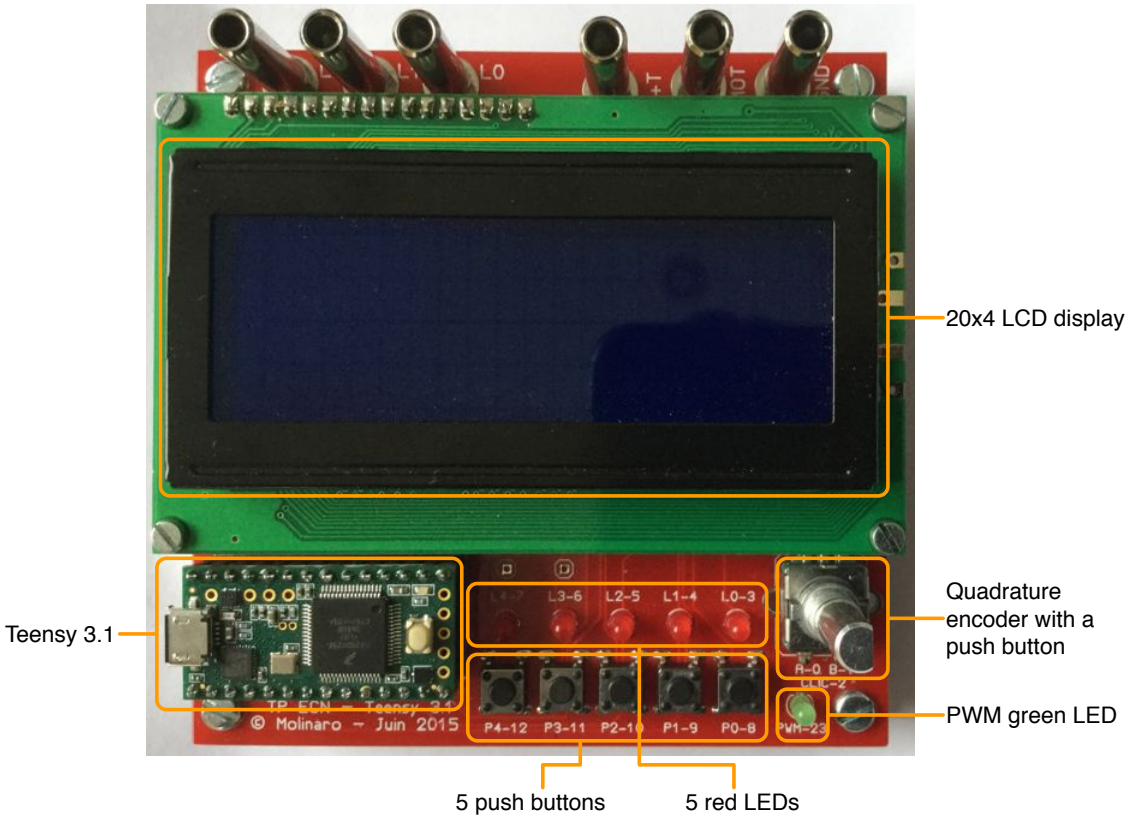
The labs board put together the following systems:

- A 20 colums, 4 lines LCD screen;
- 5 push buttons;
- 5 red LEDs;

[1] Surface Mounted Device

- A Quadrature encoder with a push button;
- A PWM[2] output / analog input;
- A green LED to display the PWM.

Here is a picture of the board:



The board power is supplied by the USB. Connect a USB port to the Teensy USB plug and the board switches on.

2.3 LEDs, buttons and LCD

The Trampoline port for this board includes functions to turn LED on and off, to read the buttons and to write to the LCD. These functions work as the functions available on an Arduino:

void pinMode(*pin***,** *mode***)** sets the mode of a pin. *pin* is the number of the pin and *mode* can be `INPUT`, the pin is an input, `INPUT_PULLUP`, the pin is an input and a resistor pulls the pin up to 3.3V (logical 1), `OUTPUT`, the pin is an output.

[2]Pulse Width Modulation

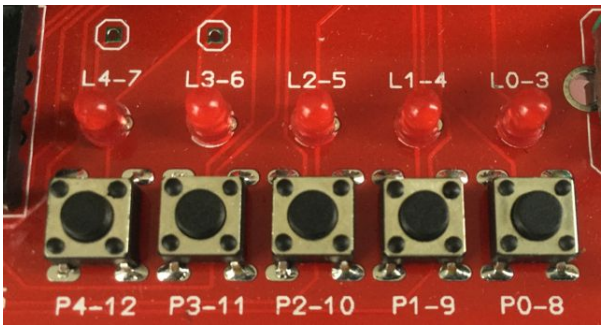
uint8 digitalRead(*pin*) returns the logical level of a pin: HIGH (logical 1) or LOW (logical 0). *pin* is the number of the pin. If the pin is programmed as an input, the value is what is set on the external pin. If the pin is programmed as an output, the value is what was written to the pin.

void digitalWrite(*pin*, *state*) set the logical level of a pin *or* enable or disable the pullup. *pin* is the number of the pin, `state` is the logical level: `HIGH` (logical 1) or `LOW` (logical 0). If the pin is programmed as an output, the state is set on the external pin. If the pin is programmed as an input, a `HIGH` state enables the pullup, a `LOW` state disables the pullup.

The LCD is accessed by using the LiquidCrystalFast library which is compatible with the LiquidCrystal library. You can get the documentation here.

2.3.1 LEDs

If you look just above the LEDs, you see labels `Lx-y`. `L` means **L**ED, `x` is the identifier of the LED and `y` is the number of the Teensy pin used to control the state of the LED.



To use a LED, you need to:

1. Program the corresponding pin, `y`, as an output;
2. Set the logical level to `HIGH` to turn the LED on;
3. Set the logical level to `LOW` to turn the LED off;

Typically, pin direction programming is done in the `main` function before starting Trampoline. For instance the following source code turns on LED 0 (pinMode and digitalWrite on pin 3) before to start Trampoline by calling the StartOS function.

```
int main()
{
    pinMode(3, OUTPUT);     /* LED L0-3 pin is set as an output */
    digitalWrite(3, HIGH); /* Turn L0-3 on                      */

    /* Start Trampoline in the default appmode */
    StartOS(OSDEFAULTAPPMODE);
```

```
    return 0;
}
```

2.3.2 Pushbuttons

As for LEDs, pushbuttons are labelled with `Px_y` where `P` stands for **P**ushbutton, `x` is its number and `y` is the number of the Teensy pin used to read the state of the pushbutton.

When pushed, a pushbutton connects the corresponding pin to the ground producing a logic 0 state (LOW). When not pushed, the pin is floating. i.e. it is not connected to anything.

Tu use a pushbutton, you need to:

1. Program the corresponding pin, `y`, as an input with pullup enabled;
2. Read the pin, `y`, to be informed of the state of the pushbutton:
   - if the returned value is `HIGH` the button is not pushed;
   - if the returned value is `LOW` the button is pushed;

The following example reads button `P0_8` to start Trampoline in a specific appmode.

```
int main()
{
    pinMode(8, INPUT_PULLUP);  /* Button P0_8 pin is set as
                                  an input with pullup enabled */
    if (digitalRead(8) == LOW) /* The button is pressed        */
    {
      /* Start Trampoline in the testAppmode appmode */
      StartOS(testAppmode);
    }
    else                       /* otherwise */
    {
      /* Start Trampoline in the default appmode */
      StartOS(OSDEFAULTAPPMODE);
    }
    return 0;
}
```

2.3.3 LCD

As already stated, the LCD is driven by the LiquidCrystalFast library. This library is written in C++. An object of type LiquidCrystalFast is instantiated with the connection pins as arguments. The LCD is connected to pins 18,17,16,15,14 and 19. So a LiquidCrystalFast object must be instantiated as a global variable as follow:

```
/*
 * A LiquidCrystalFast object is instantiated.
 * Pin numbers are per Teensyduino specification
 */
LiquidCrystalFast lcd(18,17,16,15,14,19);
```

In the main function, the LCD is initialized by using the following instruction:

```
  lcd.begin(20, 4);
```

20 is the number of columns and 4 the number of lines. Once the LCD has been initialized, the following methods can be used:

lcd.setCursor(x**,** y**)** to put the cursor at column x and line y;

lcd.print(*something***)** to write *something* on the LCD;

lcd.println(*something***)** to write *something* on the LCD and go at the beginning of the next line.

Chapter 3

Lab 1 – Understanding fixed priority scheduling

3.1 Goal

The goal of this lab is to become familiar with OSEK/VDX applications development process and with Trampoline and to understand how fixed priority scheduling works. We will also see Hook Routines and Events. Trampoline is a Free Software implementation of the OSEK/VDX specification. Trampoline includes an OIL compiler which allows, starting from an OIL description, to generate OS level data structures of the application. In addition to the OIL description, the developer must provide the C sources of tasks and ISRs implementing the application.

Get the lab1 starting source files. Go at http://www.irccyn.ec-nantes.fr/~bechenne/trampoline-labs/ and get the first archive. Once downloaded it expands to a lab1 directory.

3.2 Starting point

Go into the lab1 directory. There are 2 files:

lab1.oil the OIL description of the lab1 application.

lab1.cpp the C++ source for the lab1 task and hook routines (C++ is needed because of LiquidCrystalFast library).

Edit the lab1.oil and look at the `TRAMPOLINE_BASE_PATH` attribute (in OS > BUILD attribute). It should point to the directory where Trampoline is installed. Set it to: `/usr/local/tp_tempsreel_imta/trampoline`.

You should also modify the `LDFLAGS` lines describing the location of the libraries used

by the cross compiler. Set the first line to:

`-L/usr/local/tp_treel_imta/gcc-arm-noe-eabi-7-2018-q2-update/arm-none-eabi/lib/thumb/v7e-m`

and the second one to:

`-L/usr/local/tp_treel_imta/gcc-arm-noe-eabi-7-2018-q2-update/lib/gcc/arm-none-eabi/7.3.1/thumb/v7e-m`

lab1 is a very simple application with only 1 task called `a_task`. `a_task` starts automatically (`AUTOSTART = TRUE { ... }` in the OIL file). Look at the OIL file and the C source file.

To compile this application, go into the lab1 directory and type:

```
goil --target=cortex/armv7/mk20dx256/teensy31\
--template=/usr/local/tp_tempsreel_imta/trampoline/goil/templates lab1.oil
```

The `--target` option gives the target system (here we generate the OS level data structures of Trampoline for a `cortex` core with the `armv7` instruction set, a `mk20dx256` micro-controller and the board is a `teensy`). The `--templates` option indicates to Goil where to find the template files used to generate the C code of the configuration of the kernel. The OIL file gives the names of the C source files (with `APP_SRC` for a C file or `APP_CPPSRC` for a C++ file) and the name of the executable file (with `APP_NAME`).

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

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

Continue the build process by typing:

```
./build.py
```

The application and Trampoline OS are compiled and linked together. The target file is named `lab1_exe`.

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

```
./build.py burn
```

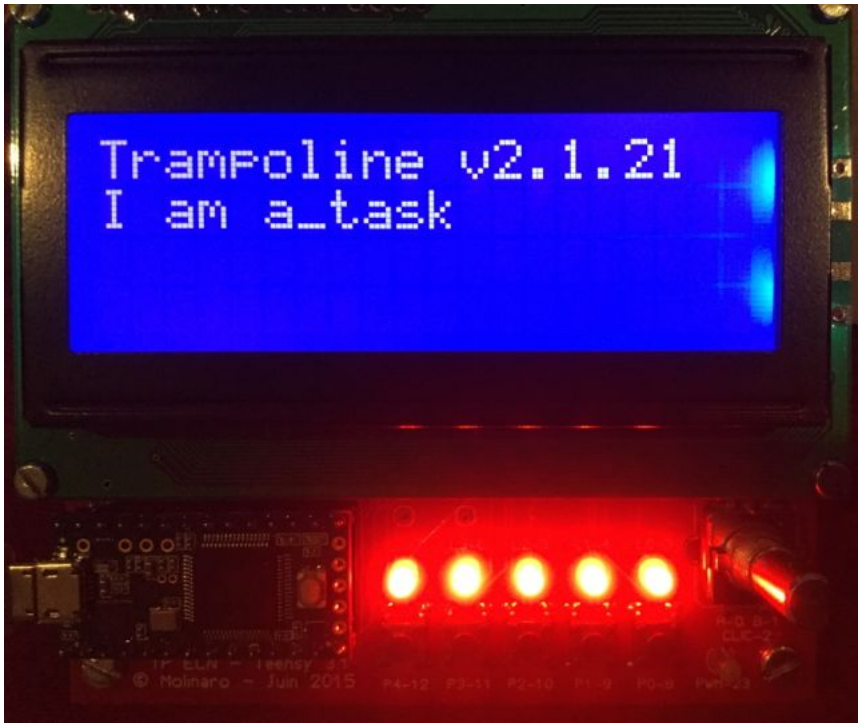
The following message is displayed:

```
Teensy Loader, Command Line, Version 2.0
Read "lab1_exe.hex": 12660 bytes, 4.8% usage
Waiting for Teensy device...
 (hint: press the reset button)
```

Press the reset button of the Teensy. Teensy loader uploads the binary to the flash of the micro-controller and displays:

```
Found HalfKay Bootloader
Read "lab1_exe.hex": 12660 bytes, 4.8% usage
Programming.............
Booting
```

The program starts and the following message should be displayed on the LCD (the first line corresponds to the execution of the main and the second line corresponds to the execution of task a_task). In addition, task a_task turns on all the LEDs.



3.2.1 A word about memory sections

AUTOSAR defines a way to put objects (constants, variables and functions) in memory sections in a portable way[1]. For that, a set of macro are used along with a generated file: MemMap.h. Functions should be declared with the `FUNC` macro, variables with the `VAR` macro, constants with the `CONST` macro and pointers to variables, pointers to constant, constant pointers to variable and constant pointers to constant with `P2VAR`, `P2CONST`, `CONSTP2VAR` and `CONSTP2CONST` respectively. Sections are opened and closed with a macro definition and the inclusion of the `tpl_memmap.h` file. For instance:

```
#define APP_Task_my_periodic_task_START_SEC_VAR_32BIT
#include "tpl_memmap.h"
VAR(int, AUTOMATIC) period;
VAR(int, AUTOMATIC) occurence;
#define APP_Task_my_periodic_task_STOP_SEC_VAR_32BIT
#include "tpl_memmap.h"
```

defines variables `period` and `occurence` in the variables section of task `my_periodic_task`.

```
#define APP_Task_my_periodic_task_START_SEC_CODE
#include "tpl_memmap.h"
TASK(my_periodic_task)
{
  ...
```

[1] memory section declaration is not part of the C standard

```
  TerminateTask();
}
#define APP_Task_my_periodic_task_STOP_SEC_CODE
#include "tpl_memmap.h"
```

defines the task `my_periodic_task` in the code section of task `my_periodic_task`. `goil` generates the sections for tasks according to the description.

3.3 OS system calls and tasks

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: `task_0` and `task_1`.*

- *add the declaration of both tasks in the OIL file; `task_0` should have priority 1 and its AUTOSTART attribute should be set to FALSE; `task_1` should have priority 8 and its AUTOSTART attribute should be set to FALSE;*
- *in the cpp file, you should:*
  - *declare both tasks with the DeclareTask keyword,*
  - *provide the body of both tasks: each task prints its name on a line of the LCD and then terminates,*
  - *and, lastly, modify the body of task `a_task` to activate `task_0` and `task_1` (in this order).*

Before to execute the resulting application, draw a schedule. Your diagram must clearly show activation, execution and termination of each job. Then execute the application and check the correctness of your diagram.

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 *Replace the call to `TerminateTask` by a `ChainTask(task_1)` at the end of task a task. 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 3 *Chain to `task_0` instead of `task_1`. 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.*

3.4 Extended tasks and synchronization using events

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 5 *Modify the application of Question 1:*

- *set priority of* `task_0` *to 8;*
- *add two events,* `evt_0` *and* `evt_1`*:*
  - `evt_0` *is set by task* `a_task` *to task* `task_0`
  - `evt_1` *is set by task* `a_task` *to task* `task_1`
- *modify the body of the tasks:*
  - *task* `a_task` *activates* `task_0` *and* `task_1` *then sets* `evt_0` *and* `evt_1` *before to terminates.*
  - *task* `task_0` *and* `task_1` *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 bodies of the task, for instance writes to the LCD or LEDs to ease the correctness checking). Comment on the differences with the previous application.

Question 6 *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 7 *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.

Go to http://www.irccyn.ec-nantes.fr/~bechenne/trampoline-labs and download the lab2 package.

4.2 First application

This application implements a periodic task that reads push button P0_8 of the board every 100ms. When button P0_8 is pushed, led L0_3 is turned on. When button P0_8 is released, led L0_3 is turned off.

Question 8 *The* `readButton` *function implements a small state machine to convert the state of the 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 timer will not be represented; it will be assumed to be negligible). Your diagram should show pressing and releasing events of button P0_8 as well as the state of led L0_3. Build and execute the application to verify your diagram.*

Modify the application: when P0_8 is pushed, a processing is triggered. This processing is performed by activating a job of task `t_process` (priority 3) that, on odd executions, displays the message *"processing triggered"*, and, on even executions, clears the LCD.

Question 9 *Draw a schedule of the application. Your diagram should show pressing and releasing events of of button P0_8 as well as the state of the LCD (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 L0_3 each time it executes and t2 toggles LED L1_4.

Question 10 *Draw a schedule of the application that show the 20 first states of the LED. What is the global period of the system?*

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

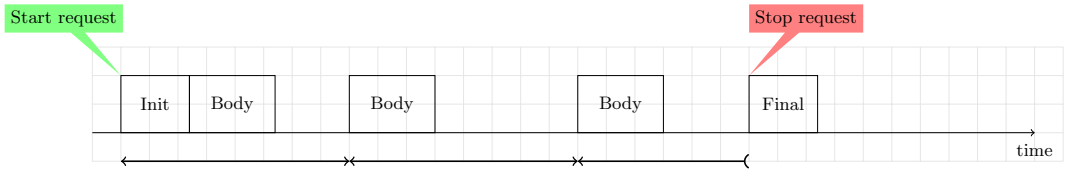
Question 11 *What is the maximal value that can be used for the* `TICKSPERBASE` *variale to fulfill the application requirements? How are the alarms configured to fulfill the application requirements?*

4.4 Third application

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

Question 12 *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.*

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



Question 13 *Modify the application to take the new requirements into account. Your implementation will use only basic tasks. 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 14 *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.

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

[2] The ShutdownOS service shutdowns the operationg 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.

In your application, each time P0_8 is pressed, P1_9 must be pressed within 2 seconds. In this case, you print the time between the two occurrences. Otherwise, an error message is displayed. If 2 or more P0_8 are got within 3 seconds from the first one they are ignored.

Question 15 *Specify this behaviour with a state machine. What is happening if the timeout occurs just after P1_9 has been pressed 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 4 periodic tasks. Each periodic task manages a LED. The chase effect is done by using alarms with a time shift between them.

When P0_8 is pressed, the chase stops. When P1_9 is pressed, the chase resumes from the states where it was stopped. When P2_10 is pressed, the chase direction changes (even if it is stopped).

Question 16 *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).

Get the lab3 package directory from http://www.irccyn.ec-nantes.fr/~bechenne/trampoline-labs.

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`, with a job activated at system startup (`AUTOSTART = TRUE`) that never ends. 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 100ms[1]. This periodic task increments the global variable `activationCount`. If `activationCount` is odd, `val` is incremented, otherwise it is decremented.
- a periodic task `displayTask` that runs every second and prints on the standard output `val` and `activationCount`.

Question 17 *Before programming the application, gives the expected sequence of values for val. Design, program, and run the application. Does it conform to the expected one? Decrease gradually the period of task* `periodicTask` *and observe the sequence of values of val. Comment.*

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.

[1] a counter gets a tick every 10ms

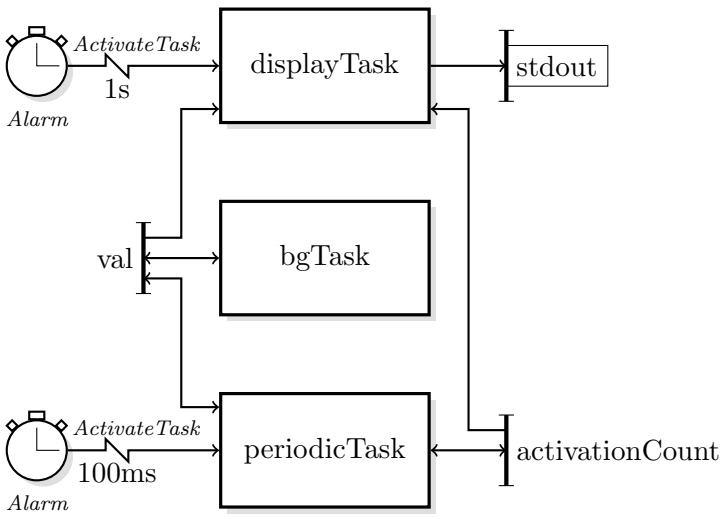


Figure 5.1: Application diagram

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

The OIL compiler (goil) 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 as long as all other data structures. These structures are commented.

Question 18

- *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?*

To observe the impact of the priority ceiling protocol, use the following function to observe the priority of the jobs during their execution. The code is provided in the file `lab3.c`.

```
void displayIdAndCurrentPriority()
{
  TaskType id;
  GetTaskID(&id);
  if (id >= 0)
  {
    tpl_priority prio = (tpl_dyn_proc_table[id]->priority) >> PRIORITY_SHIFT;
```

```
    lcd.print("Id=");
    lcd.print(id);
    lcd.print(",␣Prio=");
    lcd.print(prio);
  }
}
```

And you have to add the following line at start of your C file:

```
#include "tpl_os_task_kernel.h"
```

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 19

- *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 20 *Modify the OIL file: remove the resource and set the priorities so that no task can be preempted. Draw a schedule of the system.*