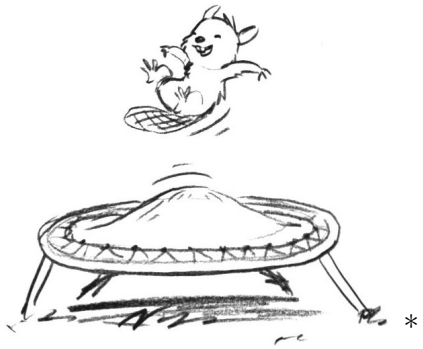
Trampoline Training

⋆



*

Jean-Luc Béchennec, IRCCyN

2016-2017

Note: All the software and documents are stored at `http://www.irccyn.ec-nantes.fr/~bechenne/trampoline`

1 Goal

The goal of this training is to become familiar with OSEK/VDX applications development process and with Trampoline. 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 of the application. Trampoline runs on many hardware platforms and we will use it on the Cortex-M4 STM32F4 Discovery board. If you have not installed Trampoline yet, get the Trampoline Package and read the install document.

The source code is located in the `labs/labs_stm32F4_discovery` directory.

*`http://fancyferret.deviantart.com/art/Trampoline-Beaver-166211545`, CC BY-NC 3.0 license, Josef Ek.

2 The board

We are going to use a demo board made by ST, the STM32F4 Discovery, with a Cortex M4 STM32F407 micro-controller. Here is a picture of the demo board:

There are 4 LEDs located below the micro-controller. LED3 is the orange one, LED4 is the green one, LED5 is the red one and LED6 is the blue one.

On the left, there is a blue button labelled `User` that can be used for user interaction.

On the right, there is a black button which is the reset button.

The LEDs and the blue button are connected to the GPIO of the micro-controller. The GPIO is initialized with the LEDs as output and the button as input. The input corresponding to the button may be configured as an external interrupt line. The initialization is done by calling the `initBoard` function. The argument of this function may be `BUTTON_NOIT` to configure the corresponding GPIO input as a normal input or `BUTTON_IT` to configure the input as an external interrupt line. In summary `initBoard(BUTTON_NOIT);` or `initBoard(BUTTON_IT);` should be put in the `main` function before starting Trampoline as shown below:

```
1 FUNC(int, OS_APPL_CODE) main(void)
2 {
3   initBoard(BUTTON_NOIT);
4   StartOS(OSDEFAULTAPPMODE);
5   return 0;
6 }
```

For the labs, functions are provided to switch on, switch off and toggle the LEDs. The unique argument is the LED identifier and should be LED3, LED4, LED5 LED6 or ORANGE, GREEN, RED, BLUE:

`void ledOn(<led>)` turns on LED `<led>`.
`void ledOff(<led>)` turns off LED `<led>`.
`void ledToggle(<led>)` toggles LED `<led>`.

A function gives the state of the `User` blue button. It returns `BUTTON_PRESSED` if the button is pressed and `BUTTON_RELEASED` if not.

`ButtonState readButton();` returns the state of the `User` blue button.

At last a function called `delay` waits for an amount of time expressed in milliseconds.

`void delay(<howManyMs>);` waits `<howManyMs>` ms.

Will will use this function to slow down the application.

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 close with a macro definition and the inclusion of the `tpl_memmap.h` file. For instance:

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

```
#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)
{
  ...
  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 Basic tasks

Go into the lab1 directory. For now, we are interested in the following 2 files:

lab1.oil the OIL description of the lab1 application;

lab1.c the C source for the lab1 task.

Edit the lab1.oil and look at the `TRAMPOLINE_BASE_PATH` attribute (in OS > BUILD attribute). `TRAMPOLINE_BASE_PATH` is set to `"../../.."`. If you move around the lab1 directory you will have to update this attribute.

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/armv7em/stm32f407/stm32f4discovery` ...
... `--templates=../../../goil/templates lab1.oil`

`goil` is the OIL compiler. It parses the OIL file and produces a set of C files. The `--target` option gives the target system. `cortex` is a group of 32-bit RISC ARM processor, `armv7em` is the instruction set of the target, `stm32f407` is the micro-controller from ST-Microelectronic and `STM32F4-Discovery` is the board.

`cortex/armv7em/stm32f407/stm32f4discovery` is a path inside the `machines` where specific source file for the target are stored. The `--templates` option gives the path to the directory where the oil files templates are stored. The OIL file gives the names of the C source files (with `APP_SRC` and the name of the executable file (with `APP_NAME`).

This generate python scripts for the application too. It has to be done only once. If you change something in the OIL file or in your C file, you do not need to rerun the goil compiler by hand because the scripts will run it when needed. Then type:

```
./build.py
```

The application and Trampoline OS are compiled and linked together. To load the application on the target, type:

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

The application may or may not start :-). Press the reset button if it does not start.

In this application, there is only one task called `a_task` which switches LED3 on.

```
1 TASK(a_task)
2 {
3   ledOn(LED3);
4   TerminateTask();
5 }
```

4 OS system calls and task launching

4.1 Task activation and scheduling

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

Go into the `lab2` directory.

In `lab2.oil` and `lab2.c`, 2 tasks have been added: `task_0` (priority 1) and `task_1` (priority 8). `task_0` toggles LED4 on and `task_1` toggles LED5 on. Task `a_task` activates `task_0` and `task_1`. All statements are separated by a busy-wait loop on the button so that by pressing the button we can control the execution. Examine the OIL and the C files.

Compile and execute. Why does `task_1` execute before `task_0` whereas it has been activated after?

4.2 Task chaining

The `ChainTask()` system call allows to chain the execution of a task to another one. This is roughly the same thing as calling ActivateTask and TerminateTask at the same time.

Replace the call to `TerminateTask` by a `ChainTask(task_1)` at the end of task `a_task`. What is happening?

Chain to `task_0` instead of `task_1`. What is happening?

Test the error code returned by `ChainTask` and correct your program to handle the error. `ChainTask` may return the following codes:

E_OS_ID the target task does not exist;

E_OS_RESOURCE the calling task holds a resource;

E_OS_CALLEVEL not called from a task;

E_OS_LIMIT too many activations of the target task.

4.3 Pre-task and Post-task hooks

Hook routines are used to insert application functions inside the kernel. Hook routines are called by the kernel when a particular event happens. The Pre-task hook is called when a task goes into the running state. The Post-task hook is called when a task leaves the running state. Hooks are useful for debugging purpose.

There are two boolean attributes in the OS object of the OIL to use Pre-task and Post-task hooks:

```
1  OS config {
2    STATUS = EXTENDED;
3    PRETASKHOOK = TRUE;
4    POSTTASKHOOK = TRUE;
5
6  ...
```

When these hooks are used, the user have to write a hook functions:

```
1 FUNC(void, OS_CODE) PreTaskHook()
2 {
3   ...
4 }
```

for the pre-task hook and

```
1 FUNC(void, OS_CODE) PostTaskHook()
2 {
3   ...
4 }
```

Go into the `lab3` directory and play with the application which is in it. It is the same application as in `lab2` with the use of `delay` instead of busy-wait for the blue button. Pre and post task hooks are used to switch a led corresponding to the running task on.

4.4 Extended tasks and synchronization using events

Go into the `lab4` directory.

This application has 2 tasks : `a_task` which is an extended task and `task_0` which is a basic task. Unlike a basic task, an extended task may wait for an event. A task is extended because it has at least one event declared in the OIL file.

Look at the OIL file and at the C file. Task `a_task` activates task `task_0` then goes into an infinite loop where it waits for event `ev_0` and activate task `task_0` again. When `a_task` runs, LED3 is switched on. When `task_0` runs, LED4 is switched on. The application should run infinitely.

Question 1 *Draw the Gantt diagram of the execution.*

Question 2 *However The application stops. What is happening ? Correct the OIL file to have a proper behavior.*

For the following application, we will use `WaitEvent`, `GetEvent` and `ClearEvent`.

Question 3 *Extend the previous application by adding 1 task: `task_1` (priority 1) and 1 event `ev_1`. `a_task` activates `task_0` and `task_1` and waits for one of the events. When one of the events is set, `a_task` activates the corresponding task again.*

5 Alarms, periodic tasks and ISR2

5.1 First application, basic use of alarms

Go into the lab5 directory.

Alarms are periodic activities. They are used to build periodic tasks. Alarms may activate a task or set an event. The application in the lab5 directory is a simple one that blinks the LED6 with a 500ms period. The `SysTick` is every 1ms. So the period of the alarm is 250ms. Examine the `lab5.oil` file and the `lab5.c` file. The underlying counter `SystemCounter` have a `TICKSPERBASE` attribute. This attribute is the number of

`SysTick` needed to increment the counter by one. By default it is set to 1. Pre and Post tasks hooks are used. Add a `delay(50);` juste after `ledToggle(BLUE);` in order to see the execution of the task.

Question 4 *Increase the `TICKSPERBASE` to 25 and change the `ALARMTIME` and `CYCLETIME` of the alarm `blink_blink` to keep the same period.*

Question 5 *Program an application using 2 periodic tasks, `t1` and `t2`. `t1` switches LED6 on and `t2` switches LED6 off. Both tasks have the same period (1s) but have an offset so that LED6 is on during 150ms.*

5.2 Second application, starting and stopping alarms

Go into the lab6 directory.

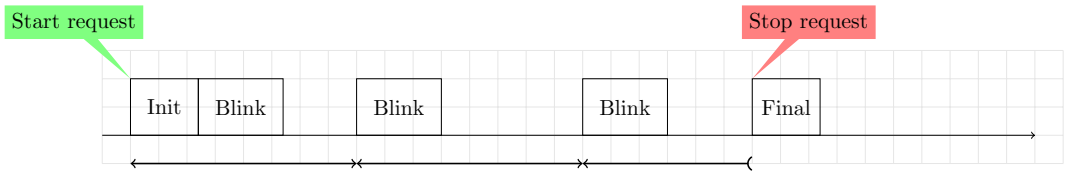
This application reads the blue button using a periodic task that is activated every 50ms. If the button is pressed, the LED6 is toggled.

Using services `GetAlarm`, `SetRelAlarm` and `CancelAlarm`, build an application with the following requirements:

- Alarm `blink_blink` activates task `blink`. This alarm is not an `AUTOSTART` one.
- Task `blink` toggles LED4.
- If the button is pressed and alarm `blink_blink` is not yet started, start the alarm with a period of 100ms.
- If the button is pressed and alarm `blink_blink` is started, cancel the alarm.

5.3 Third application

Requirements change. Now blinking needs an Init code (runs once when the alarm is started) and a Final code (runs once when the alarm is stopped). This corresponds to the following diagram:



Question 6 *Modify the application to take the new requirements into account. Use 3 basic tasks. Init and Final blink their own LED once (use delay to see the execution).*

Question 7 *Same question but with only one extended task.*

5.4 Chase

Program a chase[2] with a 0.5s period. To do it, 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 the blue button is pressed, the chase direction changes.

5.5 Using an ISR2

Instead of polling the blue button we are going to use and interrupt to trigger an ISR2. Go into the lab7 directory. Use the application as a starting point to replace button polling by ISR2 in the chase application.

6 Shared object access protection

6.1 Goal

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. We will see different ways to prevent this wrong behavior by using resources (standard and internal) or other solutions (preemption and priority).

6.2 Application requirements

The application has 3 tasks and 2 **volatile** global variables: `val` and `activationCount` as shown in figure 1:

- a background task called `bgTask`, active at start (`AUTOSTART = TRUE`) and that never ends. In an infinite loop this task increments then decrements the global variable `val`. This task has a priority equal to 1.
- a periodic task called `periodicTask`, priority 10, that runs every 100ms. This periodic task increments the global variable `activationCount` which is initialized to 0 at start. Then if `activationCount` is odd, `val` is incremented, otherwise it is decremented.
- a periodic task `displayTask`, priority 20, that runs every second. If `val` is inside interval $[-1; 2]$, LED4 is switched on and LED5 is switched off. Otherwise, LED5 is switched on and LED4 is switched off.

Describe the application in OIL and program it in C.

[2] chenillard in French

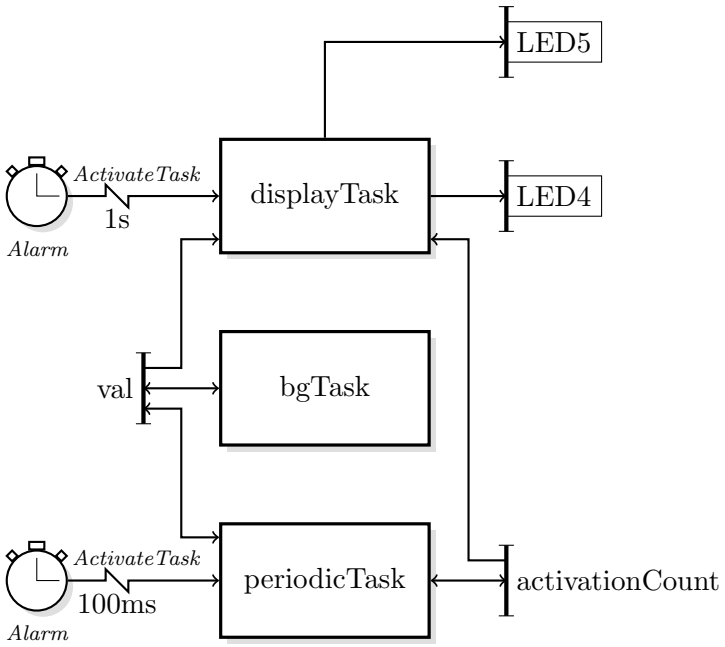


Figure 1: Application diagram

Question 8 *Does the behavior correspond to what you expect ? Why ?*

6.3 Global variable protection

Question 9 *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.

Question 10 *What priority will be given to the resource ?*

The OIL compiler (goil) generates many files in the directory bearing the same name as the oil file (less the .oil suffix). Among them 3 are interesting:

- `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 11 *For each task, find the priority computed by goil and the identifier. Is is the same as defined in the OIL file? if not is it a problem?*

Question 12 *What is the priority of the resource? Is it compliant with the PCP rule?*

6.4 Protection with an internal resource

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

Question 13 *What happens ? Why ?*

Modify the task `bgTask`: instead of infinite loop, use a `ChainTask` to the `bgTask` (ie: the task chains to itself).

Question 14 *What happens ? Explain.*

7 Adding a service into Trampoline

In this part we are going to see how to add semaphore services into Trampoline. We want two services : `SemWait` to lock a semaphore and `SemPost` to unlock a semaphore.

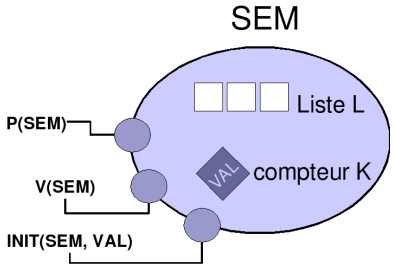


Figure 2: A semaphore object

7.1 Adding a semaphore object

Let's start with the data.

The first thing to do is to provide an object type for semaphores in C. We need a `struct` with four members. `token` is the current number of token available. `size` is the number of tasks waiting in `waiting_tasks`. `index` is write index in `waiting_tasks`. `waiting_tasks` is a ring buffer to store the tasks waiting for the semaphore.

```
typedef struct {
  uint32      token;
  uint32      size;
  unit32      index;
  tpl_task_id waiting_tasks[TASK_COUNT];
} tpl_semaphore;
```

`TASK_COUNT` is computed by goil and is equal to the number of tasks in the application.

The second thing is to provide an object type for semaphores in the OIL description. This is done in the `IMPLEMENTATION` part of the OIL description. Normally the OIL standard does not allow to create new object types. This has been relaxed in goil.

```
SEMAPHORE [] {
  UINT32 TOKEN;
};
```

This declares a new object type : `SEMAPHORE` with an `UINT32` attribute, `TOKEN`. The `[]` means multiple instances of semaphore can be used. Now, semaphores can be declared in the OIL file:

```
SEMAPHORE sem1 { TOKEN = 3; };
SEMAPHORE sem2 { TOKEN = 1; };
```

The third thing is to write a template that will generate C source code with a `tpl_semaphore` instance for each `SEMAPHORE` object in the OIL file. To do that, we have to provide a template directory hierarchy as in `goilv2/templates`. This hierarchy is put in the directory where the application source files are.

Since we generate code, the hierarchy is `goilv2/code`. Our code will be embedded in the `tpl_app_config.c` and `tpl_app_define.h`.

The templates are `custom_app_config.goilTemplate` and `custom_app_define.goilTemplate` respectively.

In the first one we have to:

1. generate semaphore object identifiers. A semaphore object identifier has the `SemType` type;
2. generate semaphore objects. A semaphore object has the `tpl_semaphore` type;
3. generate a semaphore table indexed by semaphore object identifiers. Each element of this table is a pointer to the corresponding semaphore object.

7.2 Semaphore services

Services are described in configuration OIL files. To add a service, we must provide a description of the service. This can be done in the OIL file of the application as follow :

```
1  APICONFIG semaphore {
2    ID_PREFIX = OS;
3    FILE = "tpl_os_semaphore_kernel";
4    HEADER = "tpl_os_semaphore";
5    SYSCALL SemWait {
6      KERNEL = tpl_sem_wait_service;
7      RETURN_TYPE = StatusType;
8      ARGUMENT sem_id { KIND = CONST; TYPE = SemType; };
9    };
10   SYSCALL SemPost {
11     KERNEL = tpl_sem_post_service;
12     RETURN_TYPE = StatusType;
13     ARGUMENT sem_id { KIND = CONST; TYPE = SemType; };
14   };
15 };
```

`APICONFIG` is the root object to define a set of services related to a new object. Here we define an `APICONFIG` for semaphores. goil generates identifiers for services. Identifiers are prefixed by a section name. For instance, operating system services are prefixed by OS and communication services are prefixed by COM. Here we choose to use the OS prefix: `ID_PREFIX = OS;`

The `FILE` attribute allows to list the files where the C kernel function are defined. As many files as needed may be listed. The `HEADER` attribute allows to list the files where the datatypes and constants are declared. As many files as needed may be listed.

The `SYSCALL` attribute is used to define a service. The name, here `SemWait` and `SemPost`, is the service name as seen by the application. `KERNEL` is the corresponding kernel function. `RETURN_TYPE` is the type of variable returned by the service and `ARGUMENT` is the name, type and kind of argument. As many arguments as needed may be listed (almost).

The corresponding C source code must be provided in files `tpl_os_semaphore_kernel.h`, `tpl_os_semaphore_kernel.c` and `tpl_os_semaphore.h`.

At last, the template `api.goilTemplate` is modified to add the following template code:

```
1  if exists SEMAPHORE then
2    if [SEMAPHORE length] > 0 then
3      let APIUSED += APIMAP["semaphore"]
4    end if
5  end if
```