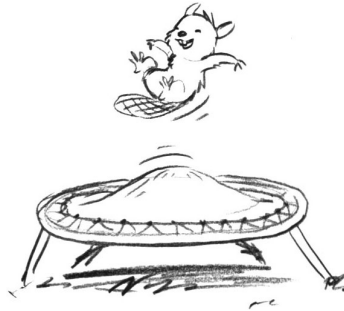


Trampoline Training



CC BY-SA 3.0



May 22, 2015

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.

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** or **LED6**:

`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 way¹. 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`,

¹memory section declaration is not part of the C standard

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 #define APP_Task_my_periodic_task_START_SEC_VAR_32BIT
2 #include "tpl_memmap.h"
3 VAR(int, AUTOMATIC) period;
4 VAR(int, AUTOMATIC) occurrence;
5 #define APP_Task_my_periodic_task_STOP_SEC_VAR_32BIT
6 #include "tpl_memmap.h"
```

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

```
1 #define APP_Task_my_periodic_task_START_SEC_CODE
2 #include "tpl_memmap.h"
3 TASK(my_periodic_task)
4 {
5     ...
6     TerminateTask();
7 }
8 #define APP_Task_my_periodic_task_STOP_SEC_CODE
9 #include "tpl_memmap.h"
```

defines the task `my_periodic_task` in the code section of task `my_periodic_task`.

3 Basic tasks

Go into the `lab1` directory. There are 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 -t=thumb2/cortex-m4/STM32F4-Discovery lab1.oil
```

`goil` is the OIL compiler. It parses the OIL file and produces a set of C files. The `-t` option gives the target system. `thumb2` is the instruction set of the target, `cortex-m4` is the micro-controller core and `STM32F4-Discovery` is the board.

`thumb2/cortex-m4/STM32F4-Discovery` is a path inside the `machines` directory and

in the `templates` directory. 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 a Makefile 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 make will run it when needed. Then type:

```
make
```

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

```
make 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.

Unlike a basic task, an extended task may wait for an event. In the OIL file, set the priority of `task_0` to 8 and add 2 events `evt_0` and `evt_1`. `evt_0` is used by `task_0` and `evt_1` is used by `task_1`. `a_task` activates `task_0` and `task_1` then sets `evt_0` and `evt_1` and terminates. `task_0` and `task_1` wait for their event, clear it and terminate.

Question 1 *Write the corresponding application. Compile and execute the application. What is happening?*

Question 2 *Program an application conforming to the following requirements: The application has 2 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`. Explain how it works.

Question 3 *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.*

Question 4 *Try many priority combinations for the tasks. Explain the behavior.*