Lab #1
⋆
Getting started with OSEK/VDX
Understanding fixed priority scheduling

November 13, 2012

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

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 see some Hook Routines too 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 of the application. Trampoline runs on Unix (and on many other hardware platforms) and we will use it on this platform. If you have not installed Trampoline yet, get the Trampoline Package and read the install document.

The lab1, is located in the `trampoline/labs/lab1` directory.

2 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 and hook routines

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=posix --templates=../../goilv2/templates lab1.oil
```

Goil templates are located in `trampoline/goilv2/templates`. The `-t` option gives the target system (here we generate the OS level data structures for the posix implementation of Trampoline). 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 (ie, roughly, a file that explain to the make build system what is needed to build the application). 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 execute the application, type:

```
./elab1
```

The following message should be displayed (it corresponds to the execution of task a_task).

```
I am a task, my id is 0
```

Question 1 *The application hangs (does not exit). Why?*

Question 2 *Turn on the Pre-task hook, Post-task hook, Startup hook and Shutdown hook in the OIL file (`PRETASKHOOK = TRUE; POSTTASKHOOK = TRUE; STARTUPHOOK = TRUE;` and `SHUTDOWNHOOK = TRUE;` in OS object). Compile and execute. What is happening?*

3 OS system calls and task launching

3.1 Task activation and scheduling

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

Question 3 *Add in the OIL file two other tasks: `task_0` (priority 1) and `task_1` (priority 8). Add the corresponding functions in the C source file. `task_0` prints "I am task 0" and `task_1` prints "I am task 1". Add in the `a_task` function after the existing* *`printf` the activation of `task_0` and `task_1`. Compile and execute. Why does `task_1` execute before `task_0` whereas it has been activated after?*

3.2 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.

Question 4 *Replace the call to `TerminateTask` by a `ChainTask(task_1)` at the end of task a task. What is happening?*

Question 5 *Chain to `task_0` instead of `task_1`. What is happening?*

Question 6 *Test the error code returned by ChainTask and correct your program to handle the error.*

4 Extended tasks and synchronization using events

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

Question 8 *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 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.*

Question 10 *Try many priority combinations for the tasks. Explain the behavior.*