OSES LAB #0 – toolchain setup

Please read the assignment description carefully

Purpose of the lab

This lab is intended to help you in setting-up the work environment for carrying out the lab activities that will be proposed in the coming weeks.

Deliverables and deadlines

This lab does not foresee any deliverable; therefore, you are not expected to hand-out any lab report to the teaching assistants.

Introduction

The toolchain we will be using is composed of:

- Trampoline (https://github.com/TrampolineRTOS/trampoline.git), which is an open-source implementation of the OSEK/VDK real time operating system (more details will be provided in the coming lectures);
- C compiler for the host machine (gcc);
- Cross-complier for Arduino (avr-gcc);
- SimulIDE, a real-time electronic circuit simulator with Arduino support.

In the following two sets of instructions are provided: one for Linux Ubuntu, and one for MacOS.

In case you are using Windows 10, please refer to the following instructions about how to run Linux Ubuntu under Windows 10 (https://ubuntu.com/wsl).

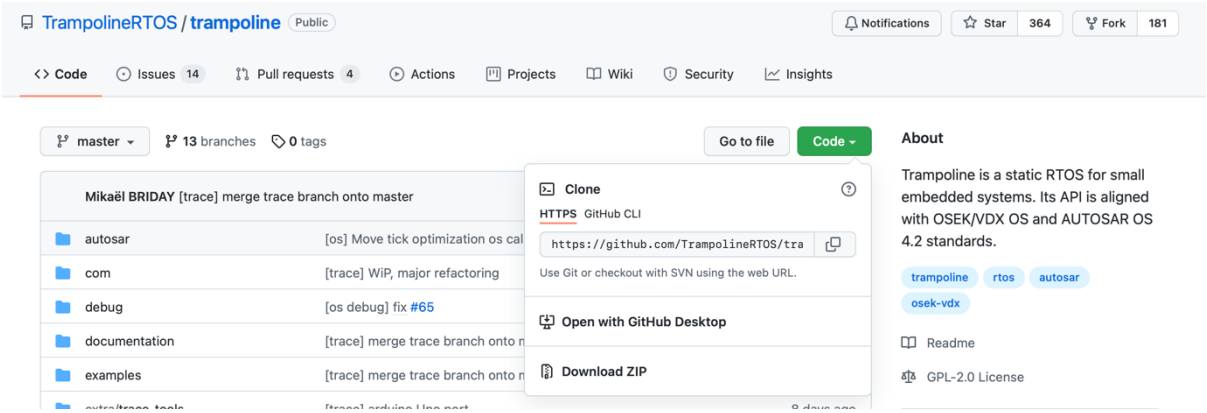
Toolchain setup under MacOS

Step #1

Download the Trampoline source code from:

https://github.com/TrampolineRTOS/trampoline

You can access the source code, by clicking on the code button as shown in the screenshot below. To simplify Steps #6 and #12, we suggest cloning the git repository rather than downloading the ZIP archive.



Step #2

Decompress the source code, and place it in any folder in your file system, for example:

```
/Users/massimoviolante/trampoline
```

Step #3

Compile the virtual peripheral simulator (viper), which is used as hardware timer simulator, and which is mandatory to run the Trampoline applications on POSIX.

```
cd /Users/massimoviolante/trampoline/viper
make
```

This will result in the viper executable file, called `viper`.

Step #4

We now have to compile the tool, called goil, needed to configure the Trampoline operating system based on the application needs (more details in the coming lectures).

```
cd /Users/massimoviolante/trampoline/goil/makefile-macosx
./build.py
```

This will result in the goil executable file, called `goil`.

Step #5

Test the set-up

Remember: each time you run a Trampoline application, you shall define the following variable, needed to make the application able to run viper.

```
export VIPER_PATH=/Users/massimoviolante/trampoline/viper
```

Now that the VIPER_PATH variable is defined, you can test an example, by first moving to its folder:

```
cd /Users/massimoviolante/trampoline/examples/posix/periodic
```

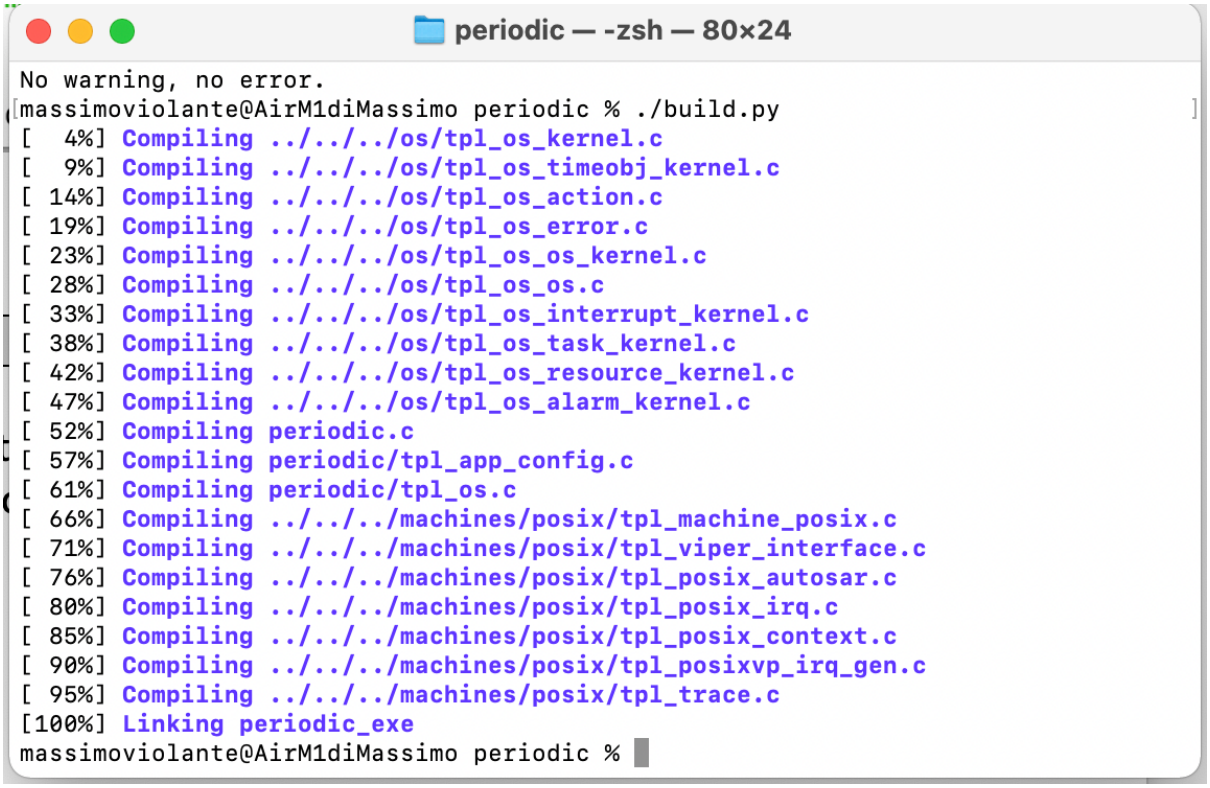
Then, you have to configure the Trampoline operating system to accommodate the needs of the application as defined in the oil configuration file (more in the coming lectures). Pay attention that the following is a single command line:

```
../../../goil/makefile-macosx/goil --target=posix/Darwin
--templates=../../../goil/templates/ periodic.oil
```

You can now build the application:

```
./build.py
```

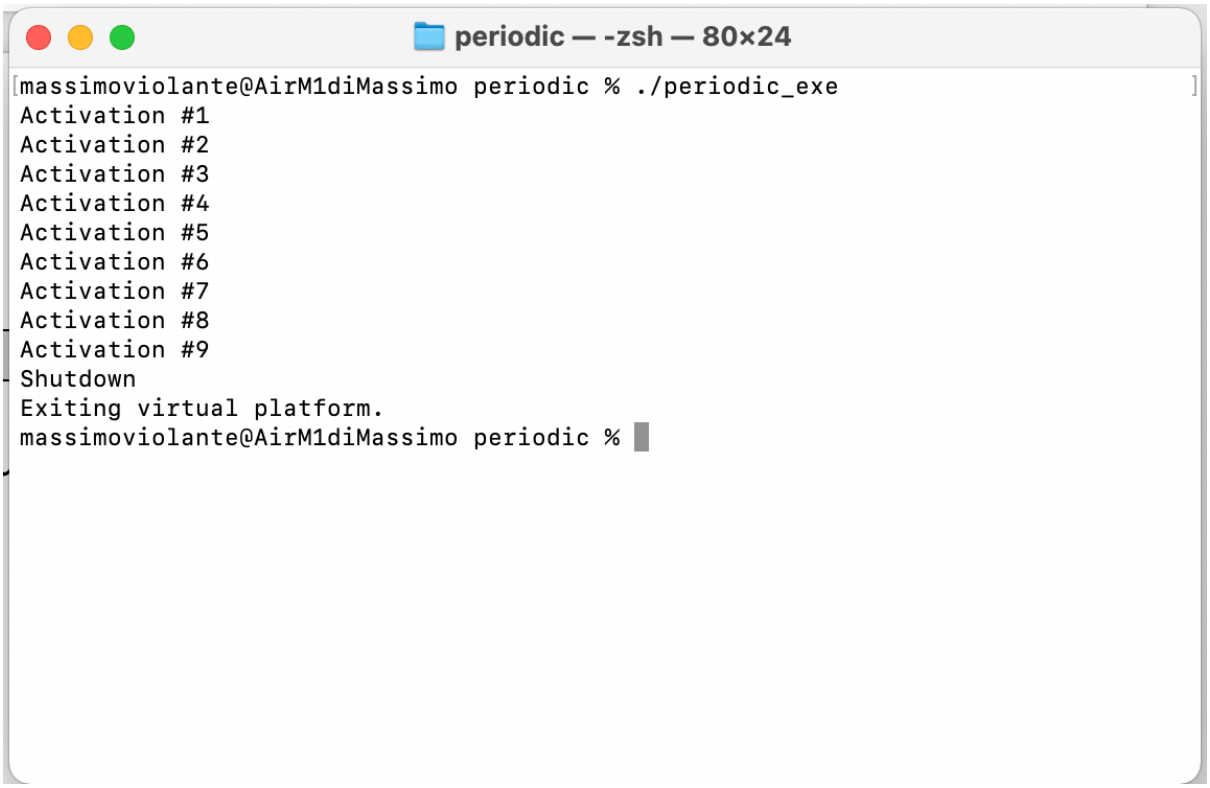
this will create the file `periodic_exe`, which is the executable of the application (`periodic.oil`) and the OSEK/VDX kernel. In case the build process is completed correctly, you shall see the following messages on screen:



You can now run the application as:

```
./periodic_exe
```

If successful you shall see the following output:



Step #6

To set-up the Arduino cross compiler, you shall perform the following actions:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew tap osx-cross/avr
brew install avr-gcc
brew install avrdude

cd /Users/massimoviolante/trampoline/
git submodule init
git submodule update machines/avr/arduino
```

Note that the `git submodule` commands above assume Trampoline has been obtained by cloning the git repository, that is, git metadata have been downloaded. You can now test if the set-up was successful:

```
cd /Users/massimoviolante/trampoline/examples/avr/arduinoUno/blink

../../../../goil/makefile-macosx/goil --target=avr/arduino/uno
--templates=../../../../goil/templates/ blink.oil

./make.py
```

The last step to be performed is to connect the Arduino Uno on the USB port and download the application.

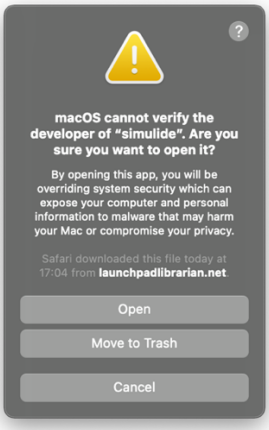
```
sudo sh -c "export AVRDUDE_PORT=/dev/ttyACM0; ./make.py flash"
```

In case the operation is completed successfully, the board LED shall start blinking with a period of about 2 seconds. Skip this part and go directly to Step #7 if you don't have an Arduino Uno board at hand.

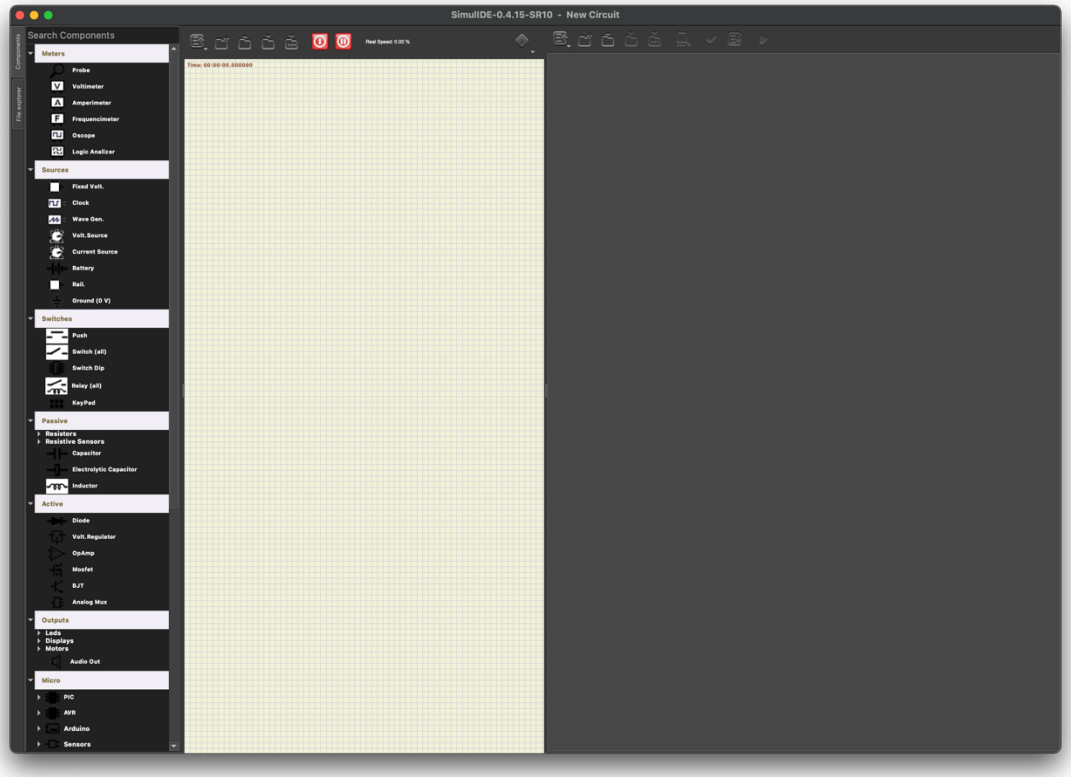
Step #7

SimulIDE for macOS can be downloaded from the project web site https://www.simulide.com/p/downloads.html. Version 0.4.15 is recommended. After downloading it, you can move the SimulIDE application anywhere in the filesystem.

The very first time you run SimulIDE, you should do so by right-clicking on it, and then choosing "Open" from the drop-down menu, instead of double-clicking on it as usual. This is because the application is not digitally signed, and hence, recent versions of macOS require an explicit, manual confirmation to override their security settings and let the application run anyway. Select "Open" again when you see the following prompt. You must do this only once because macOS will remember your choice.



SimulIDE should then display its main window and start with an empty circuit.



The SimulIDE web site also contains a comprehensive set of tutorials: https://www.simulide.com/p/blog-page.html. Of particular interest for our lab is the one on microcontrollers: https://www.simulide.com/p/blog-page_4.html. It shows how to add an Arduino microcontroller to a circuit and upload firmware into it.

The easiest way to check if SimulIDE works is to create a simple circuit with just an Arduino Uno, upload the firmware you created in Step #6 into it, and start the simulation. The onboard LED should blink with a period of about 2 seconds, as it does on the real hardware.

The right firmware image file is `trampuinoBlink.hex`, right in the main folder of the example, which is in Intel HEX format. If you are in doubt, you can find the name towards the end of the output of `make.py`:

```
./make.py
…
[100%] Linking trampuinoBlink
```

```
Generating binary trampuinoBlink.hex from trampuinoBlink
```

The difference between `trampuinoBlink.hex` and `trampuinoBlink` is that the latter is an ELF file, a file format that SimulIDE currently does not support.

Toolchain setup under Linux Ubuntu

Step #8

Download the Trampoline source code as in Step #1 for MacOS, decompress it, and place it in your file system, for example:

```
/users/massimoviolante/trampoline
```

Step #9

Compile the virtual peripheral simulator (viper), which is used as hardware timer simulator, and which is mandatory to run Trampoline applications on POSIX.

```
cd /users/massimoviolante/trampoline/viper
make
```

This will result in the viper executable file, called `viper`.

Step #10

We now have to compile the tool, called goil, needed to configure the Trampoline operating system based on the application needs (more details in the coming lectures).

```
cd /users/massimoviolante/trampoline/goil/makefile-unix
chmod u+x build.py
sudo apt-get install python
./build.py
```

This will result in the goil executable file, called `goil`.

Note: you may have to install Python. In such a case here is the command line:

```
sudo apt-get install python
```

Step #11

Test the set-up

Remember: each time you run a Trampoline application, you shall define the following variable, needed to make the application able to run viper.

```
export VIPER_PATH=/users/massimoviolante/trampoline/viper
```

Now that the VIPER_PATH variable is defined, you can test an example, by first moving to its folder:

```
cd /users/massimoviolante/trampoline/examples/posix/periodic
```

Then, you have to configure the Trampoline operating system to accommodate the needs of the application as defined in the oil configuration file (more in the coming lectures). Pay attention that the following is a single command line:

```
../../../goil/makefile-unix/goil --target=posix/linux --
templates=../../../goil/templates/ periodic.oil
```

You can now build the application:

```
./build.py
```

You can now run the application as:

```
./periodic_exe
```

Step #12

To set-up the Arduino cross compiler, you shall perform the following actions:

```
cd /users/massimoviolante/trampoline
sudo apt-get install avr-libc gcc-avr avrdude
git submodule init
git submodule update machines/avr/arduino
```

You can now test if the set-up was successful:

```
cd /users/massimoviolante/trampoline/examples/avr/arduinoUno/blink

../../../../goil/makefile-unix/goil --target=avr/arduino/uno
--templates=../../../../goil/templates/ blink.oil

./make.py
```

The last step to be performed is to connect the Arduino Uno to the USB port and download the application:

```
sudo sh -c "export AVRDUDE_PORT=/dev/ttyACM0; ./make.py flash"
```

In case the operation is completed successfully, the board LED shall start blinking with a period of about 2 seconds. Skip this part and go directly to Step #13 if you don't have an Arduino Uno board at hand.

Note: In case you are using Ubuntu under Windows 10, the serial port that is used normally to communicate with Arduino is COM5. Therefore, the command line to download the program is the following:

```
sudo sh -c "export AVRDUDE_PORT=/dev/ttyS5; ./make.py flash"
```

Troubleshooting: In case you are using Ubuntu under Windows 10, you may experience an error the first time you program the board (the programming tool times out while trying to access the board). The solution to this problem is to perform once the following operations:
1. Install the Arduino IDE;
2. Open an example program, such as Blink example from the Arduino library of examples;
3. Download the example using the Arduino IDE.

After completing these steps, the Ubuntu programing tool will start working properly.

Step #13

SimulIDE for Linux can be downloaded from the project web site https://www.simulide.com/p/downloads.html, as in Step #1. Version 0.4.15 is recommended. The "Linux 64" package works on most recent Linux distributions.

First, you should install some Qt 5 runtime libraries needed by the SimulIDE GUI. For Ubuntu 20.04, use the following command:

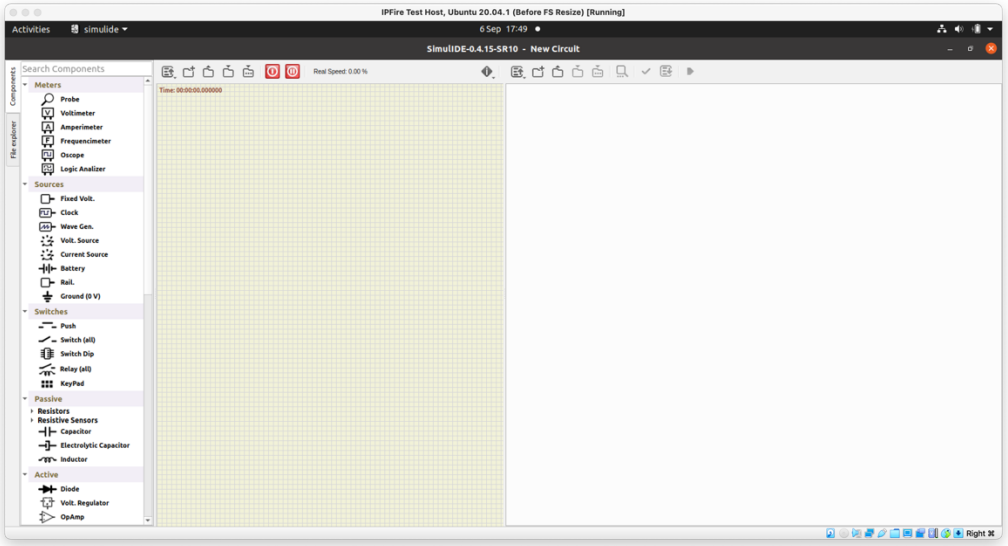
```
sudo apt-get install libqt5xml5 libqt5script5 libqt5serialport5
```

Then, after downloading SimulIDE, you shall expand the tar archive with the command:

```
tar xf SimulIDE_0.4.15-SR10_Lin64.tar
```

Within the `SimulIDE_0.4.15-SR10_Lin64` directory that this command creates, the SimulIDE executable file is `bin/simulide`. If you decide to move SimulIDE elsewhere in the filesystem (recommended), please move the whole tree rooted at `SimulIDE_0.4.15-SR10_Lin64` and not just the executable, otherwise it will no longer work.

Assuming your current directory is `SimulIDE_0.4.15-SR10_Lin64`, when you run `bin/simulide` you shall see the SimulIDE main window:



See Step #7 for additional information about SimulIDE tutorials and how to check if SimulIDE works.

Note: If you are using Windows 10, the recommended course of action is to run SimuIDE as a native Windows application rather than via WSL. Choose the "Windows 32" or the "Windows 64" version depending on your CPU architecture (32- or 64-bit). After downloading, expand the ZIP package and you will find the executable file `simulide.exe`. If you decide to move SimulIDE elsewhere in the filesystem (recommended), please move the whole tree rooted at `SimulIDE_0.4.15-SR10_Win64` and not just the executable, otherwise it will no longer work.