2 Milestone II – Install and test sensor software

The goal of this milestone is to install sensor software grove.py in Pi and test various sensors available in the lab.

2.1 Install sensor software

2.1.1 Base hat configuration

Follow "base hat configurations" instructions on the Grove website (scroll down until you see "Base Hat Configuration").

After successfully following instructions, everything should be in your `PATH` (on the Pi), so one should be able to create a separate folder (outside `~/grove.py` folder), and still be able to import the Grove libraries in your python scripts.

- https://wiki.seeedstudio.com/Grove_Base_Kit_for_Raspberry_Pi/#basic-configuration
- https://github.com/Seeed-Studio/grove.py
- https://github.com/Seeed-Studio/grove.py/blob/master/doc/README.md
- https://seeed-studio.github.io/grove.py/

The grove Python API is not only the option to interact with the Grove sensors on a Raspberry Pi. Another option is to use the Dexter Industries GrovePi library, which is available for many other programming languages as well (other than Python). For example for the buzzer sensor, both options are illustrated (with proper links to further instructions) on the wiki. Similar instructions can be found for other sensors as well.

It should be noted that the buzzer example with PWM does not work as illustrated in the wiki (only works with Raspberry 3 and below). However, the simplified buzzer example *does* work, so please **use that API**. You could also use grove_pwm_buzzer.py from here.

2.1.2 Led button demo

To check if everything is working, follow the LED button demo.

2.2 Working with Git

One of the popular programs for version control is Git. It is open source, scalable to track everything from small solo projects to complex collaborative efforts with large teams. We have created a repository for each team of this course in the KU Leuven GitLab. The idea is that you need to push each version of your Project files in this repository. Ask any Generative AI tools to generate a tutorial to set up a secure connection and use GitLab. Compare this tutorial with human generated tutorials. It is mandatory to set up git both in pi and your laptops.

2.3 Experiment with other sensors

The online documentation of the Grove sensor kit includes several tutorials or "lessons" to get acquainted with other sensors in the kit (motion sensor, temperature, light sensor, ultrasonic, ...). It is **strongly advised**

to *take all lessons* and experiment with the API of the Grove library. It is the most efficient way to get to know the available functions of the sensors, since the official API is not always clear, especially for inexperienced programmers.

2.3.1 Led Button script using arguments

Create the following python script on your laptop (`demo_0_led.py`), copy the following code into Pi and execute it.

```
#!/usr/bin/env python
import sys
import time
from grove.grove_ryb_led_button import GroveLedButton

ledbtn = GroveLedButton(5)
nr_of_events = 4
flicks_per_event = 5

if len(sys.argv) == 1:
    print("Using default parameters")
elif len(sys.argv) == 3:
    nr_of_events = int(sys.argv[1])
    flicks_per_event = int(sys.argv[2])
else :
    info = """[ERROR] Wrong number of arguments
You need the pass 3 arguments:
    (arg 1) nrOfEvents (int)
    (arg 2) nrOfFlicksPerEvent (int)
e.g.:
    $ python demo_0_led.py 4 5"""
    print(info)
    sys.exit()

print("Running demo ({0} events, {1} flicks/event)".format(nr_of_events,
                                                           flicks_per_event))
for i in range(nr_of_events):
    print("firing event {0}".format(i+1))
    for j in range(flicks_per_event):
        ledbtn.led.light(True)
        time.sleep(0.15)
        ledbtn.led.light(False)
        time.sleep(0.15)
    ledbtn.led.light(False)
    time.sleep(2)

print("Successfully finished led flicker program")
```

```
$ python demo_0_led.py <nrOfEvents> <flicksPerEvent>
```

2.3.2 Run script as a daemon

There are several ways to run a script or program as a daemon service, that is, in the background. Recollect the CLI commands you practiced in *milestone I*.

One way to execute a daemon service is by the following command on the Pi (5000 events, so this takes more than an hour to complete):

```
$ python demo_0_led.py 5000 5 > /dev/null 2>&1 &
```

Without getting too much into detail, the `> /dev/null 2>&1` part redirects the output to `/dev/null`, so all output (errors, print statements) will not be shown in the console. The "&" at the end daemonises the program. To stop this program, one has to find out the process ID (PID) of the python process, and terminate it, for instance:

```
$ kill <PID>
```

- https://linuxconfig.net/manuals/howto/how-to-find-out-the-pid-of-process-in-linux.html
- https://www.tecmint.com/find-process-name-pid-number-linux/
- https://stackoverflow.com/questions/19233529/run-bash-script-as-daemon

2.3.3 Modify a script on the Raspberry Pi

It was already discussed how to create and modify a script on your laptop, and copy it to the Pi, but you can also create and edit files on the Pi itself, either by:

- using a desktop UI service, such as VNC, and then open an editor (with a user interface) like Thonny, or
- using a text editor in the command line, such as Vim, Nano, Emacs,...

Find out how to do this, by modifying the above script; for example, by **adding** a print statement somewhere in the program, e.g., something like `print("Hello, my name is <yourname>")`. Run the script again, and check if the print statement is successfully executed.

- https://core-electronics.com.au/tutorials/basics-writing-your-first-script-with-raspberry-pi.html
- https://www.geeksforgeeks.org/linux-text-editors/

2.4 Create and run your first script

Previous sections showed how to create and edit files, and how to run them on the Pi. Let us start again from the basic LED light example, and use this code, together with inspiration from previous (code) examples, to extend this script with *new* functionality. For simple, small standalone programs (scripts) like the ones in our examples, you *could* **omit**:

- the `main()` function (the name of this function, i.e. "main", can be anything), and
- the if __name__== "_main_": part.

This "main function" workflow, and working with functions in general, *does* offer several advantages, however, especially when you are completely new to programming in general, it can be confusing at first. In the end, it is **up to you how you implement everything!** An alternative implementation for the LED demo (based on the examples in the official grove.py README):

```
import time
from grove.grove_ryb_led_button import GroveLedButton
ledbtn = GroveLedButton(5)
while True:
    ledbtn.led.light(True)
    time.sleep(1)
    ledbtn.led.light(False)
    time.sleep(1)
```

Now, create a script that has the following **features**:

- Let the pin number (5 in the example) be an argument to the program, i.e., the pin number can be configured from the command line.
- *Keep* the **while loop** of *the code example*, and turn the LED on and off for 5 seconds, resp., anddo this 3 times in a row (*TIP: use a counter*).
- Terminate the program automatically after this process is finished.

A `while True` loop is in fact an infinite loop, which is fine for our use cases, since we can always terminate the program with `Ctrl+C` and keep it running as long as we want (or terminate it programmatically after process X is finished).

The `GroveLedButton` class (imported from the grove package) takes care of more than simply connecting the LED sensor to a pin on the board (I/O). For example, it has methods to set the light on and off. This class moreover handles clicks on the button (which will mostly likely issue a print statement). This is typical behaviour for **higher-level libraries**, whereas low-level libraries ("closer to the metal/electronics"), for example, to program a connection between a PLC and a LED light, will merely focus on the I/O connection (and required additional layers for callbacks, events, ...). The level abstraction is different for each library/API/programming language, but generally speaking, "higher level" means less complexity, less work (implementation wise), but also less "freedom for the programmer". At the lowest level, we find machine code, i.e., binary (1's and 0's) code that can be executed directly by the CPU.

When you omit `#!/usr/bin/env python` at line 1 in your script, you have to use `$python <myscript>` command, instead of `$./myscript` (the latter requires making the program executable with `$ chmod +x)`. The default version of python on Bullseye is python3.