Cours de Systèmes Embarqués et Robotique
Section de Microtechnique
Printemps 2022

Practical Exercises

Practical Exercise 4: CamReg	
Title:	Setting of a distance between the robot and a visual object with an optical tracking and a PI regulator
Goal:	To program in a real case a PI regulator and a simple, but real time, image processing in order to keep a given distance between a printed sheet and the robot.
Duration:	4 hours
Support:	Files available to download listed and explained in the appendix.
Equipment:	e-puck2 robot, ChibiOS library, Python

1 Main goal

The goal of this practical session is to implement a PI regulator in order to keep a given distance between the robot and a vertical black line printed on a white sheet by using the camera to detect it and measure the distance. This will be achieved by reading one line of pixels of the camera, processing it to find the black line and compute the distance and writing and setting a simple PI regulator to control the motors.

1.1 Methodology

To achieve the main goal, we will go through the following steps:

- Understanding how to communicate with the e-puck2 throught USB or Bluetooth.
- Understanding how the camera works, what are the values returned and what the functioning conditions are.
- Implementing and verifying an algorithm to find the line and compute the distance.
- Using the distance measurement to write a PI regulator used to keep the robot at a certain distance.
- Verifying the characteristics of the regulator and correcting some implementation's problems with an ARW (Anti Reset Windup) and/or other mechanisms.

2 Setup of the project

For this practical exercise, you will have to add some libraries next to the pratical exercices folders. In the archive downloaded from Moodle, you will find the folders **lib** and **TP3_Chibi** that you are asked to put in the same folder as the previous practical exercises. Make sure to fusion the files in the **lib** folder in order to have the same folders as in the figure 1.

You will also find a folder named **Scripts_python** which contains python scripts used for this practical session and for the next. You can put it where yo want.

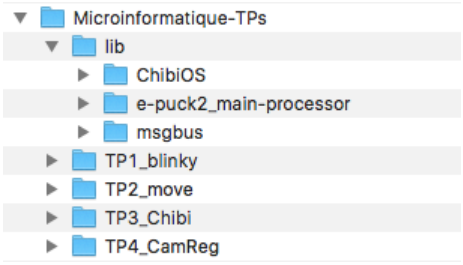

Figure 1: Folder structure you should have

For this practical exercice, you need to open the folder **CamReg** in Eclipse to open the project. The TP4 will use files from the **lib** folder. This is why you will find a linked folder inside the Project Explorer in Eclipse pointing to e-puck2_main-processor.

3 e-puck2_main-processor library

Until now, we used simple libraries for TP1 and TP2 and ChibiOS for TP3. From now, we will use the **e-puck2_main-processor** library, which is the library written for the e-puck2. It contains nearly all the drivers needed to use the functionalities of the robot and it uses ChibiOS to run. For example the files to use the I2C and the IMU in TP3 were taken from this library. You are free to look at it to see how things are done if you want, or simply to see what is possible to do with it. All the functionalities we will use in this TP come from e-puck2_main-processor. This is the library you will use for the mini-projects too.

The project you have for this practical session uses a simple makefile which sets some parameters, and then calls the bigger makefile of the e-puck2_main-processor library.

4 Different ways of communication with the e-puck2

The e-puck2 has quite complex interconnections between its components. What interests us for this practical exercise is to understand the different ways you can use to communicate with the microcontroller such that you are able to use them in your code to transmit data to the computer. The figure 2 shows you the connections between the microcontroller, the programmer and the ESP32 (Bluetooth module). You can see that you can use two different protocols with the microcontroller. One directly with the **USB** of the microcontroller and the other with the **UART3** which then goes through the programmer and the ESP32. The programmer converts it to a USB com port and the ESP32 converts it to a Bluetooth com port.

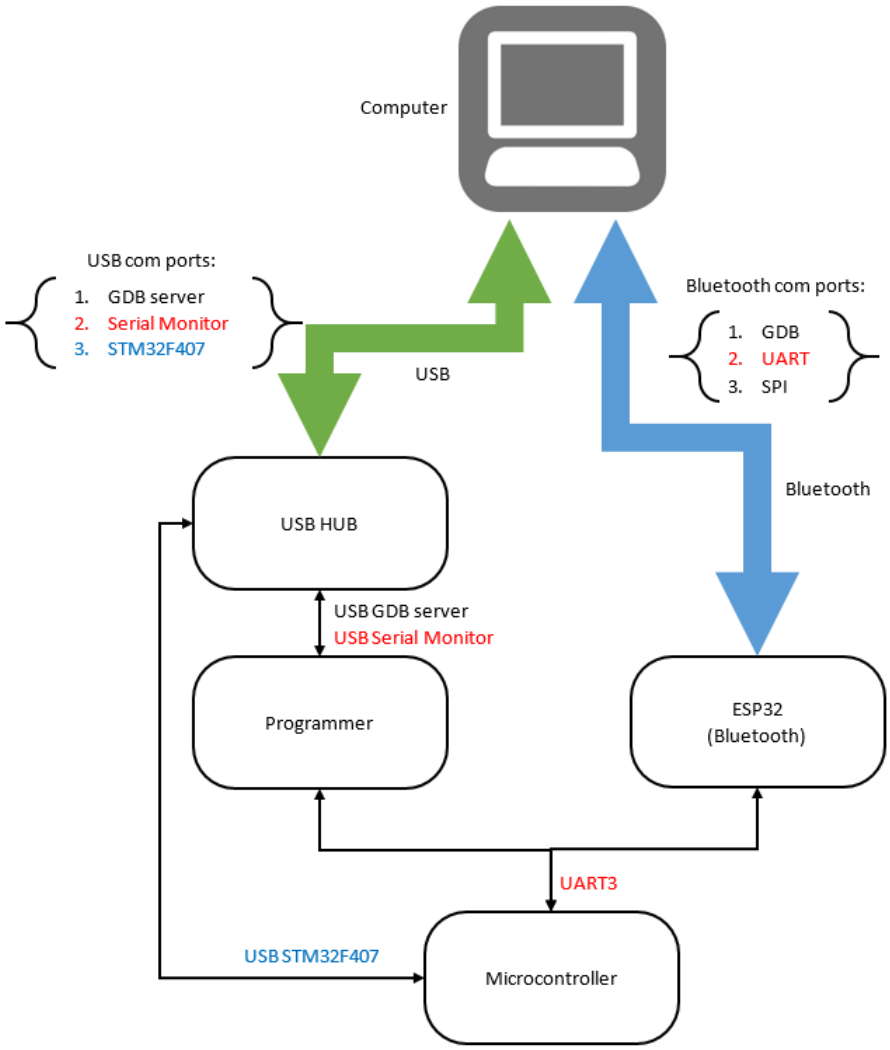

Figure 2: Communication scheme of the e-puck2's microcontroller.

So you have two distinct ways to communicate with a computer that you can use simultaneously and independently, namely the USB and the UART3.

4.1 Functions to communicate with ChibiOS

ChibiOS offers several functions to send or receive data. Thanks to the Abstraction Layer implementation in ChibiOS, you can use the same functions with the UART3 or the USB. The difference is which pointer you will use as argument in the functions. Below are some examples of communication functions :

```
//formatted print of the variable "time" to SD3 (UART3)
chprintf((BaseSequentialStream *)&SD3, "time␣=␣%d␣\n", time);

//formatted print of the variable "time" to SDU1 (USB)
chprintf((BaseSequentialStream *)&SDU1, "time␣=␣%d␣\n", time);

//sends the buffer "data" of size "size" to SD3 (UART3)
chSequentialStreamWrite((BaseSequentialStream *)&SD3, data, size);

//reads from SDU1 (USB) "nb_values" values and stores them to the buffer "data"
```

```
chSequentialStreamRead((BaseSequentialStream *)&SDU1, data, nb_values);
```

What you need to remember are the pointers **SDU1** (USB) and **SD3** (UART3) which indicate which interface you will use to communicate. The cast **(BaseSequentialStream *)** is here to convert the type of the pointer into the good one. It works without it but you will have warnings during the compilation.

4.2 Behavior of the different com ports

Another thing concerns the behavior of the RTOS when using these functions. The USB protocol being much more complex than the Serial one (UART), it implements a complete flow control. The Serial could also have a flow control but it would have taken more than two pins for the communication. So when using one of the communication functions with the USB, if the USB cannot send the data, for example when the cable is not connected, **the thread in which the function is called is put into sleep until the sending is possible.** This behavior doesn't apply to the UART because it simply sends the data, without taking care if someone on the other side is listening or not.

So simply be sure the USB cable is connected and you are reading the com port when using a communication function with the USB of the microcontroller.

If you need to use only one communication protocol, we advise you to use the **UART3**, as it gives you the possibility to use the Bluetooth or the USB without changing the code and without blocking the code if the cable is not connected.
But if you need to use two independent communication channels, then you can add the use of the USB of the microcontroller.

Table 1: Behavior of the different com ports

Communication protocol	Behavior when not connected	Must disconnect from the com port before reprogramming ?
USB (microcontroller)	**blocking**	**yes** because if we are programming the microcontroller, then it cannot control its USB
UART3 (Bluetooth and USB Programmer)	**non blocking**, no matter if connected or not	**no** because the Bluetooth and the USB connected to the UART3 are controlled by other components

5 Using the Bluetooth of the e-puck2

5.1 Pairing with the e-puck2

The e-puck2 is not visible on the Bluetooth by default. To put it into discoverable mode, you need to hold the **User** button (next to the **Reset** button) while turning ON the robot.

Then you need to go on the **Devices** pane of the **Settings** of Windows, select **Bluetooth & other devices** on the left and choose to add **Bluetooth or other device** (fig. 3). The window you will differ depending on the OS you use. The figure 3 shows the one you will have with Windows 10. Make sure the Bluetooth is turned on, otherwise you will get an error.

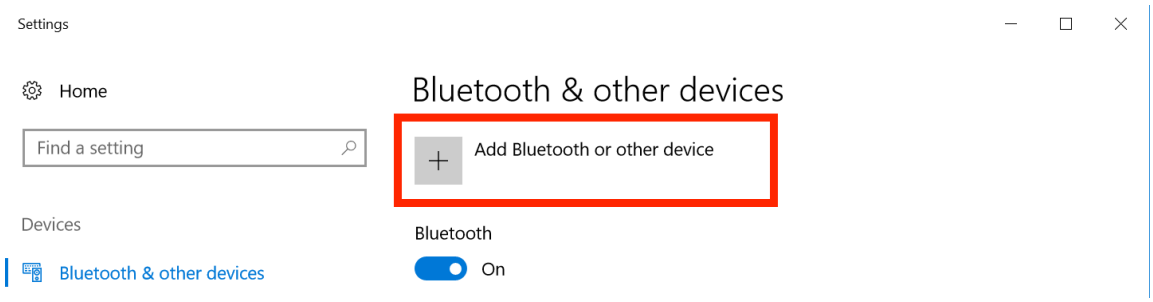

Figure 3: Window to add a Bluetooth device on Windows 10

The e-puck2 will appear with this kind of name : **e-puck2_xxxxx**, xxxxx being the number you can find on the e-puck2. During the pairing process, it is possible that the computer asks if the code shown is correct (fig. 4). As the e-puck2 doesn't have any screen, you simply need to click **Connect** and the pairing will be complete. You will maybe have to wait several seconds in order to let Windows configure the new peripheral.

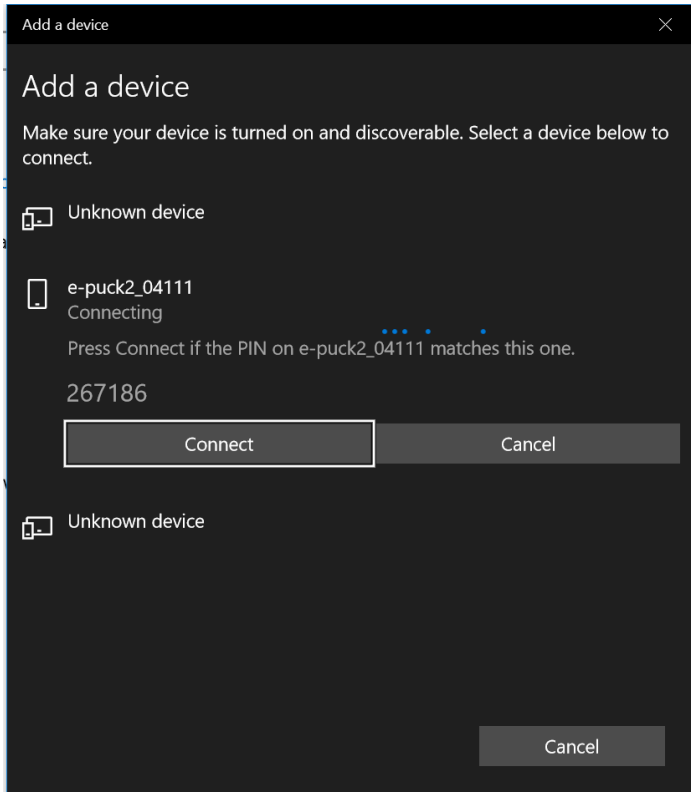

Figure 4: Final step to add the e-puck2 on Windows 10. Example with the e-puck2 4111

5.2 Finding the Bluetooth COM port

After the pairing process, you can connect to the e-puck2 by Bluetooth with a terminal program for example, like in TP3. The way you can connect to the Bluetooth is exactly the same as to connect

to the USB of the e-puck2. You just need to find the correct com port to use. You will have the choice between three new com ports (possibly six for windows 10).
To know which com port is the UART one, you need to do the following steps :

- **For Windows 10** : go down (or right) in the **Bluetooth & other devices** window and click on **More Bluetooth options** (fig. 5). Then go on the **COM ports** tab (fig. 6). It will give you a list of Bluetooth com ports. You will find easily which one is the UART one.
- **For Mac** : Open a terminal window and type the command **ls /dev/cu.***. It will simply list you all the com ports you have and you will find the one you're searching (listing 1).

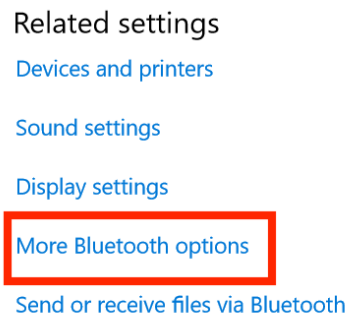

Figure 5: More Bluetooth options link on Windows 10

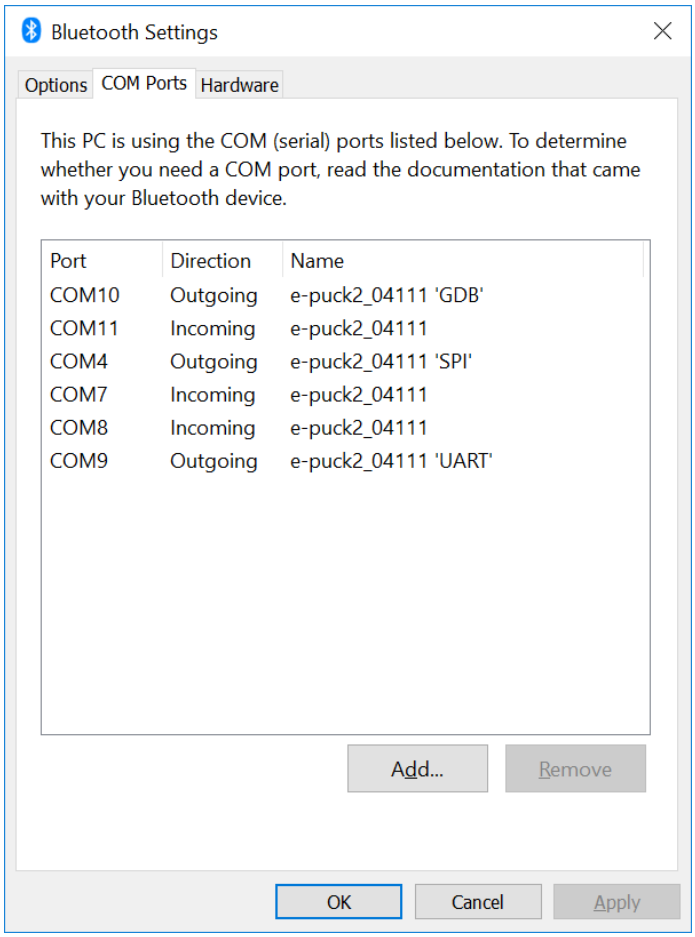

Figure 6: More Bluetooth options window on Windows 10. Shows a list of the Bluetooth com ports

Listing 1: Example of com ports on Mac with three different e-puck2 paired

```
$ ls /dev/cu.*
>/dev/cu.Bluetooth-Incoming-Port  /dev/cu.e-puck2_03958-SPI
>/dev/cu.e-puck2_00002-GDB        /dev/cu.e-puck2_03958-UART
>/dev/cu.e-puck2_00002-SPI        /dev/cu.e-puck2_04111-GDB
>/dev/cu.e-puck2_00002-UART       /dev/cu.e-puck2_04111-SPI
>/dev/cu.e-puck2_03958-GDB        /dev/cu.e-puck2_04111-UART
```

5.3 Blue led indicator

The blue led located near the USB connector has a special role. It blinks when UART3 is communicating and when no data is sent or received it remains OFF if the bluetooth is not connected and ON if the bluetooth is connected. So once connected to the bluetooth of the e-puck2, the blue led should turn on, indicating a bluetooth connection is active.

5.4 Troubleshooting with the Bluetooth

- On windows, it is possible that the terminal program gives you an error when trying to connect to the Bluetooth com port. Most of the time, this is resolved by turning off and on the Bluetooth on windows.

- Connecting to Bluetooth need more time than connecting to USB. Thus after a press on connect in the terminal program, wait about 5 seconds before clicking again thinking it does nothing.
- Remember: the Bluetooth and the USB are connected to the UART port of the microcontroller. So if the Bluetooth reception is bad, or if it is too slow, or anything else that prevents you to use the Bluetooth, don't hesitate to switch to the Serial Monitor port via USB. You will receive exactly the same things, but with a cable.
- Also note that the Bluetooth has a smaller transmission speed than USB. So if you transmit too many data too quickly, it is possible to loose data.

6 Threads in the code

The code given uses three threads.

- **PiRegulator** Thread doing the PI regulator stuff.
- **CaptureImage** Thread capturing the images with the camera.
- **ProcessImage** Thread processing the images captured by CaptureImage.

The thread **PiRegulator** runs at a given frequency. No matter if a new image has been processed or not. It doesn't wait for new distance measurements from the thread **ProcessImage**. It just uses it, even if it is the same for several iterations. This is done only because a PI regulator gives better results when it runs fast, even if the information it uses are refreshed slower.

CaptureImage is a thread that only launches a new capture, then waits for the capture to be finished. When it is the case, it releases a semaphore to tell a new image is ready and begins a new image capture.

ProcessImage is a thread used to process the image. Thus it needs to know when a new image is ready. It is done by looking at the same semaphore [1] as the thread **CaptureImage**.

The synchronization between **CaptureImage** and **ProcessImage** works like as follow :

1) **CaptureImage** is a producer and releases the semaphore when a new image is ready.

2) **ProcessImage** is a consumer and waits until the semaphore is released. When it is the case, it can take it. To be able to take the semaphore on the next iteration, it will again need to wait until the semaphore is released.

By only releasing the semaphore with the producer thread and only taking it with the consumer thread, we ensure a correct synchronization between the two threads.

Note : All the wait functions provided by ChibiOS put the concerned thread into sleep to wait. In fact every functions implying to wait on something have this behavior.

7 Camera

The camera used on the e-puck2 is a PO8030D [2]. It is a 640x480 color camera which uses the DCMI port of the microcontroller and its angle of view is about 45 degrees.

7.1 Reading images with the camera

In the code given, the camera is configured once with the following line in the **CaptureImage** thread:

```
//Takes pixels 0 to IMAGE_BUFFER_SIZE of the line 10 + 11 (minimum 2 lines because reasons)
po8030_advanced_config(FORMAT_RGB565, 0, 10, IMAGE_BUFFER_SIZE, 2, SUBSAMPLING_X1, SUBSAMPLING_X1);
dcmi_enable_double_buffering();
dcmi_set_capture_mode(CAPTURE_ONE_SHOT);
dcmi_prepare();
```

It means we will store two lines of the image (starting from the 10th line from top) taken from the camera, each line being IMAGE_BUFFER_SIZE long (which is 640 pixels for this TP). We need to read at least two lines because of how the camera's library is written.
So we will have an image of 640x2 pixels and we will only read one line. This way we emulate a linear camera of 640 pixels.

The camera could take images of 640x480 pixels maximum but because of limited space in ram, the library limits the size to a total of 19200 bytes. Knowing the images are in a **RGB565** format, it means each pixel takes two bytes. So we can capture maximum images of 9600 pixels.
The dcmi functions are here to configure the driver on the microcontroller side when **po8030_advanced_config()** configures the camera.

Then the following functions are called to start an acquisition (must be done for each image) and to release a semaphore to tell the thread **ProcessImage** a new image is ready :

```
//starts a capture
dcmi_capture_start();
//waits for the capture to be done
wait_image_ready();
//signals an image has been captured
chBSemSignal(&image_ready_sem);
```

Then in the Thread **ProcessImage**, the following functions are used to wait until a new image is ready by using the same semaphore.

```
//waits until an image has been captured
chBSemWait(&image_ready_sem);
//gets the pointer to the array filled with the last image in RGB565
img_buff_ptr = dcmi_get_last_image_ptr();
```

Another important thing to notice is that we configure the DCMI driver to use double buffers to capture the images. This means we can process an image and capture another one in the same time.

With these functions, you can read the pixels stored inside the **img_buff_ptr** pointer which acts like an array.

7.2 Image format

A lot of image formats exist. Here, we use the **RGB565** format. It is a format that uses a 5 bits value for the red, a 6 bits value for the green and a 5 bits value for the blue. The total makes 16 bits, so each pixel is coded with a 16 bits information describing the intensity of each color. You can see on figure 7 the binary representation of the RGB565 format.

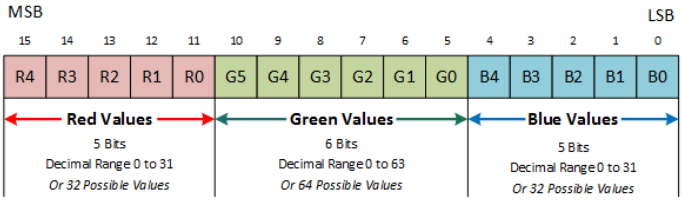

Figure 7: RGB565 binary representation. Taken from `http://henrysbench.capnfatz.com/henrys-bench/arduino-adafruit-gfx-library-user-guide/arduino-16-bit-tft-rgb565-color-basics-and-selection/`

For the practical session, you will need to extract the color of your choice to only use one color information for the line detection. As the line is black and the sheet is white, the choice of color should not change the result a lot. But it is sure you will have more information if you use the green for example, as it is coded with 6 bits instead of 5 for the others.

In the code, **img_buff_ptr** is an **uint8_t** buffer. This means you will read your 16 bits pixel through two uint8_t values in big-endian format. So be careful. In the loop you will write to extract one color information, **img_buff_ptr** will have a size two times bigger than the size of the buffer you will use to store one color value.

8 Sending the data to the computer

In this practical exercise, we will use a python script to plot the lines of pixels taken with the e-puck2. You can use the following function on the e-puck2 side to send the one-line images to the computer :

```
//sends the data buffer of the given size to the computer
SendUint8ToComputer(uint8_t* data, uint16_t size);
```

Then on the computer side you need to run the python script **plotImage.py**. You have two possibilities to run this script: you can take the precompiled script (only for windows 64bits) or you can directly launch the python script if you have python3 installed on your computer.

You can refer to the **readme.txt** located into the **Scripts_python** folder provided for more informations about how to install python and the needed libraries or how to use the precompiled version.

Task 1:

In the thread **ProcessImage**, use the code provided to read images from the camera and write a simple loop to extract one color of your choice from the images (you need to do bits manipulations). Then send them to the computer to visualize them (You need to send them relatively slowly if you don't want to loose bytes. For example you can send only one image over two).
Place the robot in front of the sheet with the black line. How does the line appear on the plot ? Is it normal or not ?

How do the values of the plot change when the camera looks at other things in the room ? Why does it behave like that ? (Hint : try to have something dark and something brighter in the same image)

Task 2:

Measure the time used to take a photo with the camera in the thread **CaptureImage**. You can use the function **chVTGetSystemTime()**. It is already used to compute the sleep duration in the thread **PiRegulator** for example. It is a measure that uses the system tick of ChibiOS, so it counts in milliseconds. But as we don't need to be more precise, we can use it instead of a timer.

What happens to the capture time if you cover the camera with your hand ?

Hint : You can use **chprintf()** to send some variables to the computer in order to know if your function works well or not. As you already are sending the images to the computer through the UART3 of the microcontroller with the dedicated functions showed before, you have two choices to use **chprintf()** :
1) Stopping to send the images and using **chprintf()** with **SD3** (UART3)
2) Continuing to send images and using **chprintf()** with **SDU1** (USB)

The camera is configured to send its images in a continuous way. Then on the microcontroller, we read only when we want to get an image. One drawback of this behavior is the synchronization. If we want to read an image while an image is already being streamed by the camera, then the microcontroller waits the beginning of the next image to capture it and we lose time waiting for the image to begin. This is why we have a thread dedicated in the capture of the images. Like that, we can be sure the synchronization is kept and we works with the images on another thread.

Task 3:

Add a sleep duration of 12ms into the thread **CaptureImage** to see the desynchronization. Is the new capture time coherent ?

Do not forget to remove the thread sleep in order to have a working function for the next tasks.

Task 4:

Now that you are able to read an image, you need to write a detection function to determine where is the black line and what is its width in pixels. Call this function in the thread **ProcessImage**. Verify you obtain a decent detection for distances between about 3 and 20 cm.

How to make this detection independent of the ambient light? (hint : average)

Task 5:

Find a way to convert the width in pixel of the black line to the distance in cm between the paper and the camera of the robot. Test your conversion and adjust it if necessary. The resulting function can be as simple as a division. Store your result in the variable **distance_cm** declared on top of the code.

(Hint : you can measure the distance between the robot and the sheet and send the width in pixels to the computer)

9 PI Regulator

Now you will have to write a PI regulator in order to control the motors to keep the robot at a given distance of the printed line. You can search on internet to find how to implement a simple discrete PI regulator if you have no clue on how to do it [3].

You can find in the file **pi_regulator.c** a thread named **PiRegulator** which is meant to run the PI regulator at a given frequency. At the end, you can find the functions to set a speed to the motors. For now they are commented to not move the robot until you have written the PI regulator. The motor's functions take speed in $step/s$ and not in cm/s like in TP2.

To access the distance computed in the thread **ProcessImage**, use the function **get_distance_cm**.

Task 6:

Write a function that implements a PI regulator in the file **pi_regulator.c** and call it in the thread **PiRegulator**, or write directly the PI regulator into the thread. Use the distance computed in the thread **ProcessImage** in the PI regulator and a goal distance of **10cm**. Why is **distance_cm** declared as static outside a function ? What does it mean ? Try to understand why we use a function to get the variable instead of directly accessing it.

Task 7:

Try your PI regulator with only the Proportional coefficient (Kp). Use a little Kp at first and then increase it. Don't forget to uncomment the functions to set the speed of the motors in order to apply the command speed computed in the PI regulator.

Try to find experimentally which Kp gives the best performance in your case.

What happens when Kp is too big or too small ?

Is the distance correct when you move the paper at a constant speed ? Why not ?

Note : the motor's library already limits the speed you can give to the motors to $\pm 2200 step/s$.

Task 8:

Now use the integral coefficient (Ki) in your PI regulator and play a little with the Ki coefficient.

What is the effect of Ki in the system ? Is it better when you move the paper at a constant speed ?

What happens with the integral part of the PI regulator when the robot is too far from the 10cm goal distance (or too close) ? (Hint : print the value of the integral sum).

Task 9:

Implement a simple ARW system by limiting the minimum and maximum values of the integral sum and then try to find experimentally which couple of Kp and Ki gives the best performance in your case.

Depending on your PI regulator, other mechanisms such as threshold can be used to correct other problems you could encounter. For example when the robot is very close to the goal distance, then depending on you Kp and Ki, it will always move a bit because of the noise on the camera.

Task 10:

Now reduce the frequency of the **PiRegulator** thread from 100Hz to 10Hz. What is the result with the Kp and Ki previously found ? Is it better or worse ? Why ?

Task 11:

BONUS : If you want, you can also add a simple control of the rotation of the robot in order to try to follow the black line. This way the robot will keep the goal distance with the paper and will try to be in front of the black line, even if you move the paper to the left of to the right.

References

[1] *Semaphores and Mutexes explained.* ChibiOS.org. 2017.
`http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:semaphores_mutexes`

[2] *PO8030D Datasheet.* Rev 0.6. PIXELPLUS. October 2013.
`http://projects.gctronic.com/E-Puck/docs/Camera/PO8030D.pdf`

[3] *Intuitive explanation on how to implement a PID in c. [French]* Ferdinand Piette. August 2011.
`http://www.ferdinandpiette.com/blog/2011/08/implementer-un-pid-sans-faire-de-calculs/`