Grenoble Institute of Technology
École nationale supérieure de physique, électronique, matériaux
Système Embarqué Numérique

# Embedded SoC Design & Integration using lowRISC

**Allibert Romain**
**Le Mière Antonin**
**Mackowiak Aurélien**
**Maldaner Liége**

Prof. Dr. Mounir Benabdendi

Grenoble, France
2020

# Contents

# 1 Introduction

The main objective of this project is to save images to an SC card, preprocess and plot it on a scream. The codes will then be used by another project group. A summary of this first project is presented in the Figure 1.1.

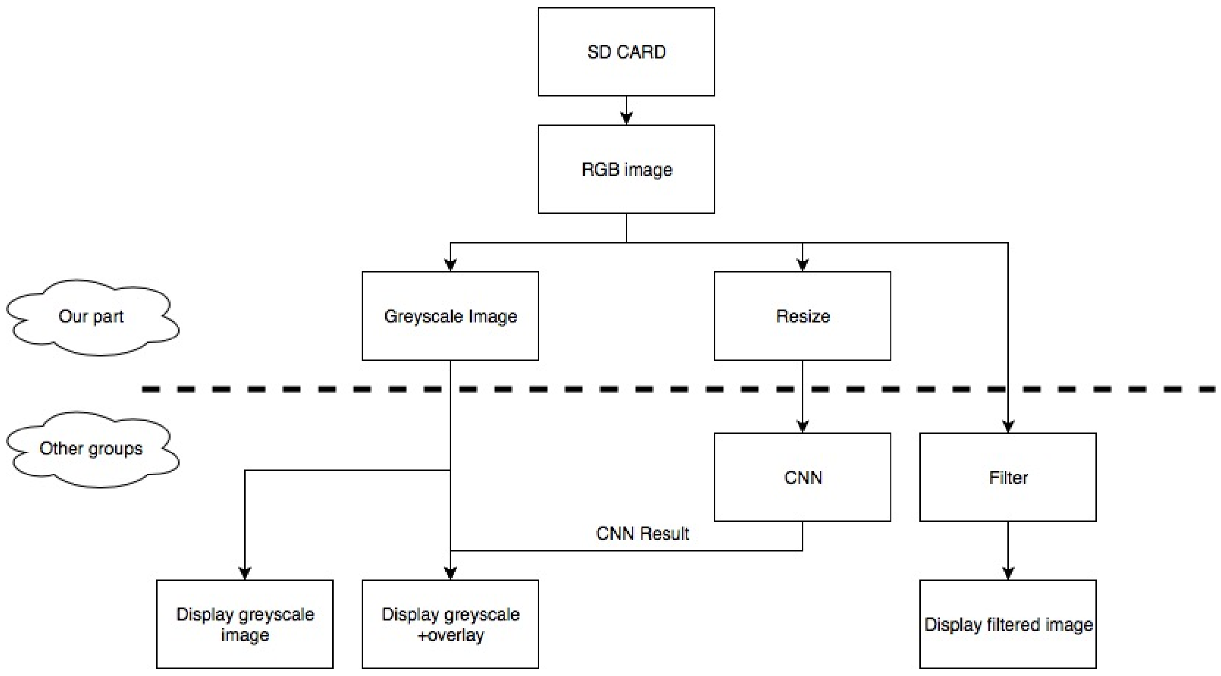


Figure 1.1: Project schematic

We need to read a JPEG file on the SD card, transform it in RGB file and treat it in RGB to output it on VGA. We also do a resizing, so it can be classified by a CNN model. The image can be output in Greyscale or not, and it is stored in RGB for the filter and the CNN.

# 2 Project 1: Getting Images from the SD card

## 2.1 Read the images from the SD card

The purpose of this part was to read images in JPEG format that were previously stored into a SD Card. To fulfill this task, we chose to to proceed step by step.

First, we wanted to understand how to implement software on the RISCV, so we decided to first open a text file and read it's content. During this process, we faced a lot's of problems mainly due to *undefined functions*. Due to our experience in coding, we knew it was usually a problem with the Makefile. We fixed the Makefile by adding the MMC file we needed and by removing the files that were compiled twice. Even with these modifications, some functions were impossible to use. We realized that most of the functions were added during the boot phase and because of this our code couldn't be compiled. Moreover, we realized that some functions were disabled by the file *ffconf.h*. Once all these modifications were made, some functions were still impossible to use or just not included in the RISCV software. To solve this issue, we added the remaining functions in the file *read_image_test1.c* in which the main function is located.

As soon as we could compile the code and read a text file, we decided to open images with an easier format : the ppm format. It's much more easier to extract the pixels from a ppm image than a JPEG one because this format is not compressed. The algorithm works that way :

1. open the file
2. get the header with the type of the image
3. if the type is correct (PPM type P3), we look for the size of the image in the header and continue with the pixel extraction
4. Pixel extraction in a tab
5. Copy in a bigger tab (for the case with multiple images to load)

The tab is filled this the pixel in the following way :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|
| R[pix0] | G[pix0] | B[pix0] | R[pix1] | G[pix1] | B[pix1] | R[pix2] | G[pix2] | B[pix2] | ... |

The next step should have been the implementation of the jpg library to compute jpg images from the SD Card. But the time went short and because our project was the first part of a bigger project and was necessary for the whole project to work, we decided to keep ppm images. The next section will explain how we managed to still work with jpg images.

## 2.2 Python script for conversion ( JPG → PPM )

Our part of the code being essential for other groups, we decided to turn to another solution, which will be easier to implement.

The problem with JPEG images comes from compression. In fact, once the file has been read, we have to decompress the image before providing it to CNN and the VGA display. Not having access to the library, it is therefore difficult to carry out the operation on the pixels.

So we opted for another option. We will write a python script that will convert a JPG image to a PPM image. The script is quite simple and uses the OpenCV image processing library.

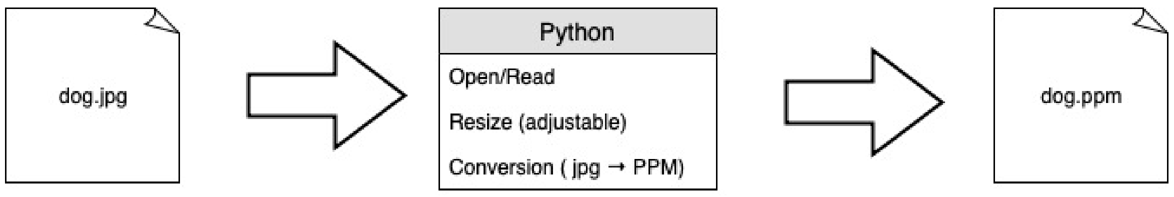


Figure 2.1: Image conversion

The script opens the image, resizes with adjustable parameters, then converts the image to the usable format.

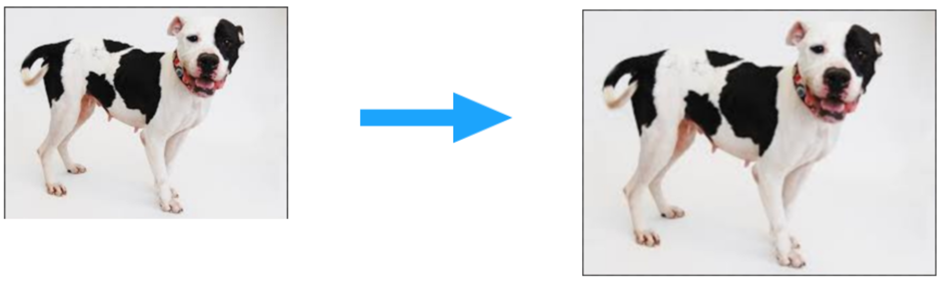


Figure 2.2: Resizing and image extraction in PPM

The problem with this script is that it only runs on x86, so we need to use the script before loading the images on the SD card. It would be more interesting to use the C libraries dedicated to image processing in the future. This could be done directly on the card.

At the end of this step, the JPG type images are all converted and are implemented on the SD card. We can therefore start to recover the information on the pixels.

## 2.3 Functions to convert from the format being read to the format on which the application operates (e.g., 640x480 to 24x24 for the CNN application.)

In this project our main goal was preprocess images to be feed into a Convolutional neural Network (CNN) algorithm. To get the right classification from the CNN, it is mandatory that the test dataset images were in the same format as the images used to train it, otherwise we can't guarantee that the images will be correct classified.

The weight and bias of the CNN model were trained with the CIFAR10 dataset, which is composed from images with size 32x32 and composed of 10 different classes: airplane, car, bird, cat, deer, dog, frog, horse, ship and truck. Those images were first resized to 24x24, and then normalized with the algorithm in Equation 2.1 below. So our dataset must pass through the same processing.

$$\begin{aligned} \mu &= \frac{1}{N}\sum m_{i,j} \\ \sigma &= \sqrt{\frac{1}{N}\sum (m_{i,j}-\mu)^2} \\ m^{'}_{i,j} &= (m_{i,j}-\mu)/max(\sigma, \frac{1}{\sqrt{N}}) \end{aligned} \tag{2.1}$$

We implement the resizing algorithm, which is basically a convolution of the image with an unitary kernel, as presented in 2.2 and the normalization, as presented in Equation 2.1.

$$\begin{aligned} kernelSizeX &= 640/24 \\ kernelSizeY &= 480/24 \\ kernel &= kernelSizeX \times kernelSizeY \\ I[0,0,channel] &= \left(\sum_{0}^{kernel} image[0:kernelY, 0:kernelX, channel]\right)/kernel \end{aligned} \tag{2.2}$$

where the kernel will flows through the image, summing all pixels, then diving it by the kernel size to create a new pixel of the resized image. In this case, 640/24 give us the float value, 26.66, so we defined it as an integer function.

To test the algorithm, we use the Figure of a Cat (class 3), showed in Figure 2.4 and the result is presented in the Figure 2.3.

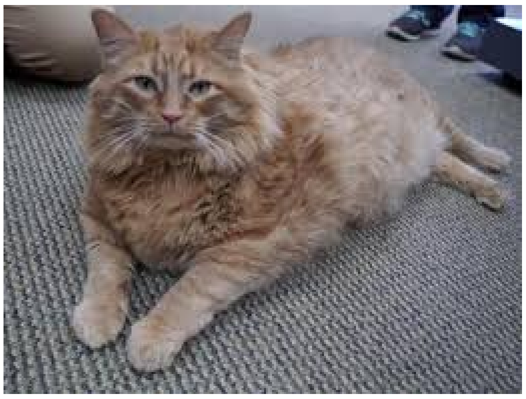


Figure 2.3: Image of a cat

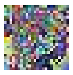
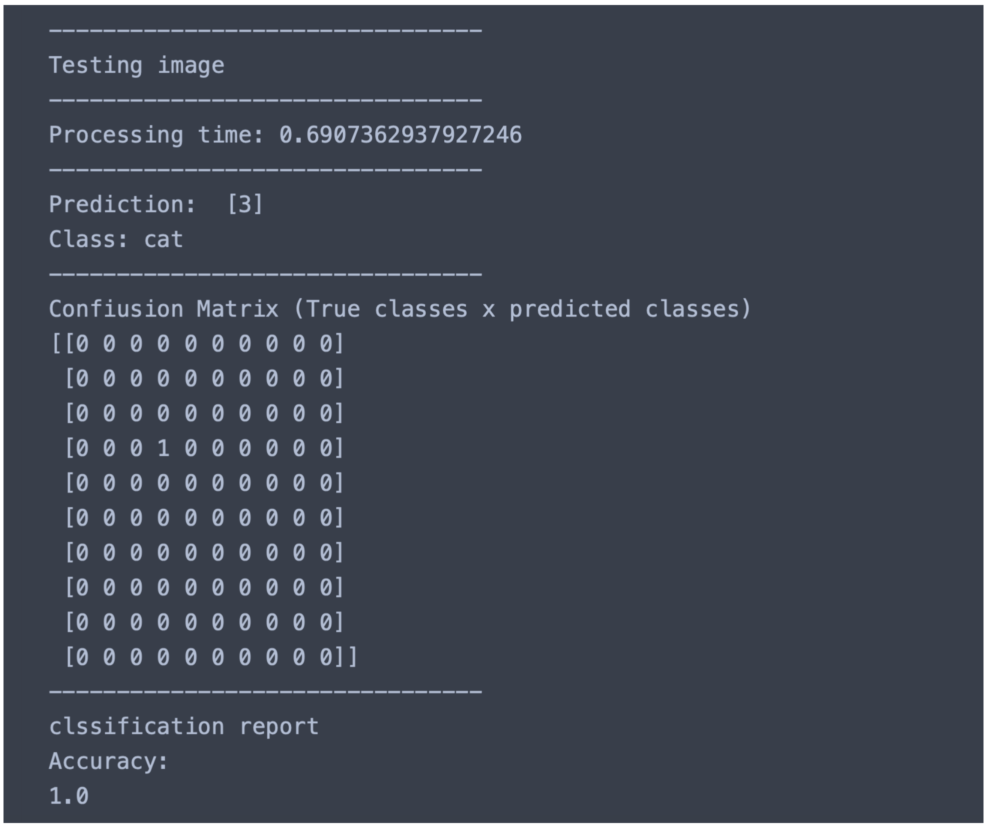


Figure 2.4: Image of a cat resized to 24

Feeding the CNN algorithm with the Figure 2.5, we get the classification result as 3, which means the class cat, as shows the Figure below. We can conclude that the resizing and normalization works correctly to a random image (that is not included into the CIFAR10 dataset).



```
--------------------------------
Testing image
--------------------------------
Processing time: 0.6907362937927246
--------------------------------
Prediction:  [3]
Class: cat
--------------------------------
Confiusion Matrix (True classes x predicted classes)
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
--------------------------------
clssification report
Accuracy:
1.0
```

Figure 2.5: CNN classification result

## 2.4 Display the image on VGA

To display the image on VGA, the first part is to recover the image generated into an array with the following organization :

$$Pixel0\ RGB,\ Pixel1\ RGB,\ \ldots \tag{2.3}$$

Once the array is stored, we can apply the RGB to Grayscale transformation. We choose a transformation who is comfortable for human eye. In this way, the green is the most important part because we are more sensitive to this color. We use this algorithm :

$$GS = 0.3 * R + 0.59 * G + 0.11 * B \tag{2.4}$$

Once the image has been converted in Grayscale, we can output it to VGA. We use the width and length of the image to output it on VGA, and fill the rest of the image with black pixels. This part is automated by the image size given with the ppm file. This field is stored and used in the code.

## 3 Conclusion

This project could improve the knowledge on RISCV, in the context of loading images from an SD card and processing them. It was also possible to plot the images in grayscale or RGB on a screen, and process them to be used into a CNN model. The codes can be find attached to the same repository as this report.

Since RISCV is an open source approach with highly potential on the processors industry, using it during the academic year was a great way to get a better knowhow on it and improve our skills.