PYNQube

Massimo Perini, Davide Toschi, Qi Zhou

Supervisor: Marco Domenico Santambrogio

University: Politecnico di Milano

Team number: xohw18-412

Short video: https://youtu.be/HGkxSjZM94A

June 30, 2018

Contents

1. Introduction

In these days, embedded systems are coming up in a big way. Connected devices, machine to machine (M2M), Internet of Things (IoT) and the penetration of Electronics in our daily lives allows to use embedded systems in very diverse scenarios. In this context a platform like PYNQ can show its value since integrates an ARM Cortex processor and a Xilinx Zynq in the same board, allowing very diverse usages of the same device in very different contexts. In this scenario we wanted to perform the resolution of a Rubik Cube using the PYNQ platform in order to show its flexibility. Our project includes several steps in order to have the cube solved. It shows how PYNQ is able to handle different tasks without any problems thanks to the different hardware components it is made of.

2. Design

In this chapter, a description of the PYNQube project implementation is provided. This project is composed by several steps executed in order to get the cube solved. Some of them have been implemented in hardware, others in software

- Image processing and elaboration: in this step we recognize the Rubik cube initial state. It has been implemented with the usage of OpenMP library, since they are integrated into the PYNQ platform. In order to speed up the process, we adopted the ZYNQ too, using the Vivado Design Suite tools by targeting the PYNQ-Z1 board (ZYNQ XC7Z020CLG400-1 part with PYNQ Preset)
- Cube solution computation: We calculated the required moves that needs to be executed with an implementation of the Kociemba two-phase algorithm. This algorithm performs two optimized searches on a tree with the Artificial Intelligence IDA* algorithm in order to find the moves.
- Moves execution: ....

2.0.1 *Hardware*

Image processing

We perform image processing in order to extract the color of the faces of the Rubik cube. Given an image as input, our algorithm transforms it into 3 array. A component of the array is the R, G or B value of a pixel. We compute the average of the values of each colored face of the Rubik cube for each color channel. In order to do such, our function works using 3 values we previously defined: the number of cells in each face, the total number of pixel transmitted. The number of pixel for each colored square is therefore the product of the first two values. Changing these values we can use the same function with different number of squares on each face. Since this operation is computationally heavy, it has been implemented using the Xilinx ZYNQ component of the PYNQ board. We adopted a stream communication since the type of computation we need to perform doesn't require to save the data and, in this way, the system become stateless and not dependent from the size of the input values. Python and Zynq communicates through the DMA memory.

Analog Hardware Design

The last step of PYNQube project is to control the rotation of cube's facets. Our goal is to show how Pynq is flexible and powerful also to control a complex external hardware. During the develpement of the circuit, we decided to use Pynq as the crux of our architetture. Thanks to PMOD port controlled by base bitstream of FPGA, we could implement an i2c bus connected to pin 2 and 6 of PMOD-A port as SDA and SCL. Pynq is the master of the bus , so it decides which data and when data has to share. There are three slaves in our system: ATmegs328. ATmegs328 is an 8 bit microcontroller with 1KB EEPROM, 2KB SRAM, 23 general purpose I/O lines, 32 general purpose working registers. The device operates between 1.8-5.5 volts. We set for each slaves a different index: 0x05, 0x06, 0x07. On the market there are several types of motor ranked by price, power consumption and accuracy, we needed a precise one with a small step angle in order to rotate cubes with fewer errors as possible, so we chose NEMA 17 Stepper Motor, a bipolar stepper. These motors have 4 wires for the 2 phases (A+ , A-, B+, B-), phase resistance of 3.5ohms and a rated current of 1A. Due to this value of current ATmega328 cannot menage the power supply of motor so we must use a driver. We opted for L298n. The L298 is an integrated monolithic circuit in a 15-lead Multiwatt. It is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external terminal can be used for the connection of an external sensing resistor. An additional supply input is provided so that the logic works at a lower voltage. The operating supply of this device is up to 48 V and the total DC current is up to 4 A. This circuit allow us to move and resolve the cube thanks to the algorithm computed from Pynq with a good accuracy and performance.

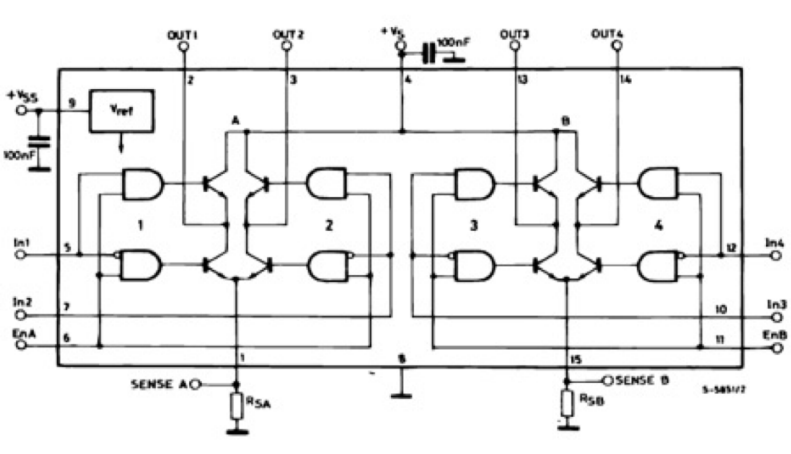

Figure 1: Schematic of L298n driver

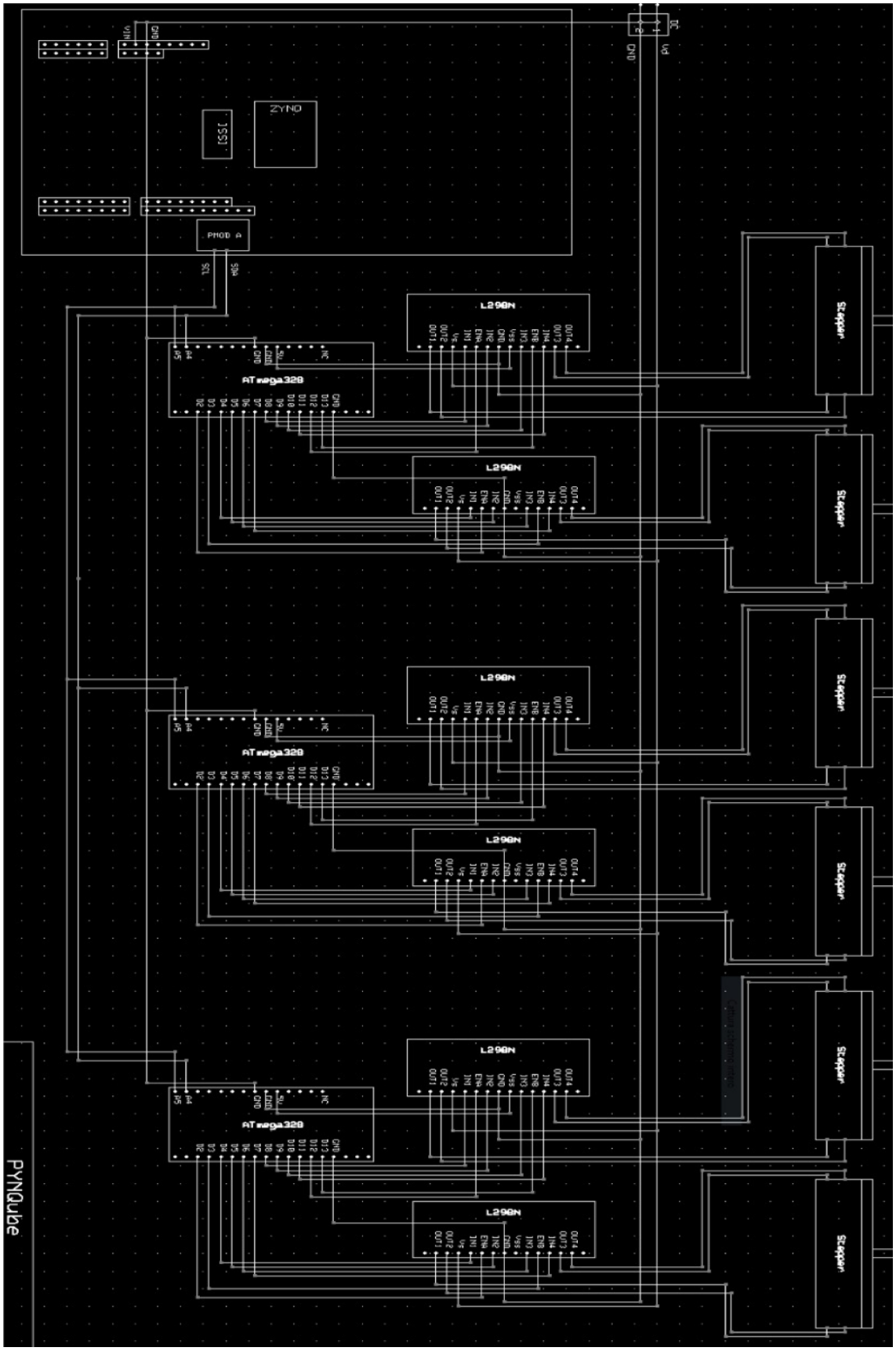

Figure 2: Analog circuit design of PYNQube

2.0.2 *Software*

We computed the Rubik Cube solution implementing the Kociemba 2-phase algorithm. Our algorithm performs two Iterative deepening A* searches in the state tree. We choose this algorithm since the memory on PYNQ is very limited. Each node is represented as a Rubik Cube object and, starting from the initial state object, the algorithm generates its children and keeps only the good ones. A good child:

- Is not redundant (did not appear in the previous levels of the tree)
- Its heuristic function value is less than the threshold

The heuristic function is the sum of two components: the first one increases with an increased depth of the search tree, in order to penalize long solutions. The second one assigns a score to the node adopting previously computed tables. We generated these tables with a breadth-first algorithm. We stored as depth 0 the goal state and we applied every different move to it. The different results will be stored as depth 1. In the next pass, then, we re-applied the moves to each state saved as depth 1, the results will be stored with a depth 2 and so on. In this way during the execution of the solver we will be able to retrieve a lower-bound of the moves required to solve the Rubik cube . Having a heuristic function that provides a lower bound estimation is a requirement of the IDA* algorithm too. Our algorithm, so, keeps going deeper until the heuristic function is greater than a certain threshold. When this happens, the algorithm try another node instead of expanding the current one. If the solution is not found, the threshold is increased and the search starts again. This is a two-phase algorithm since two different searches are performed: in the first stage we are looking for some particular states and in the second one, starting from one of these states, we can reach the solution adopting only a subset of possible moves. Please notice that this implementation is very extensible: states are able to generate their children. This means that the same search algorithm is not strictly linked with the definition of state we adopted.

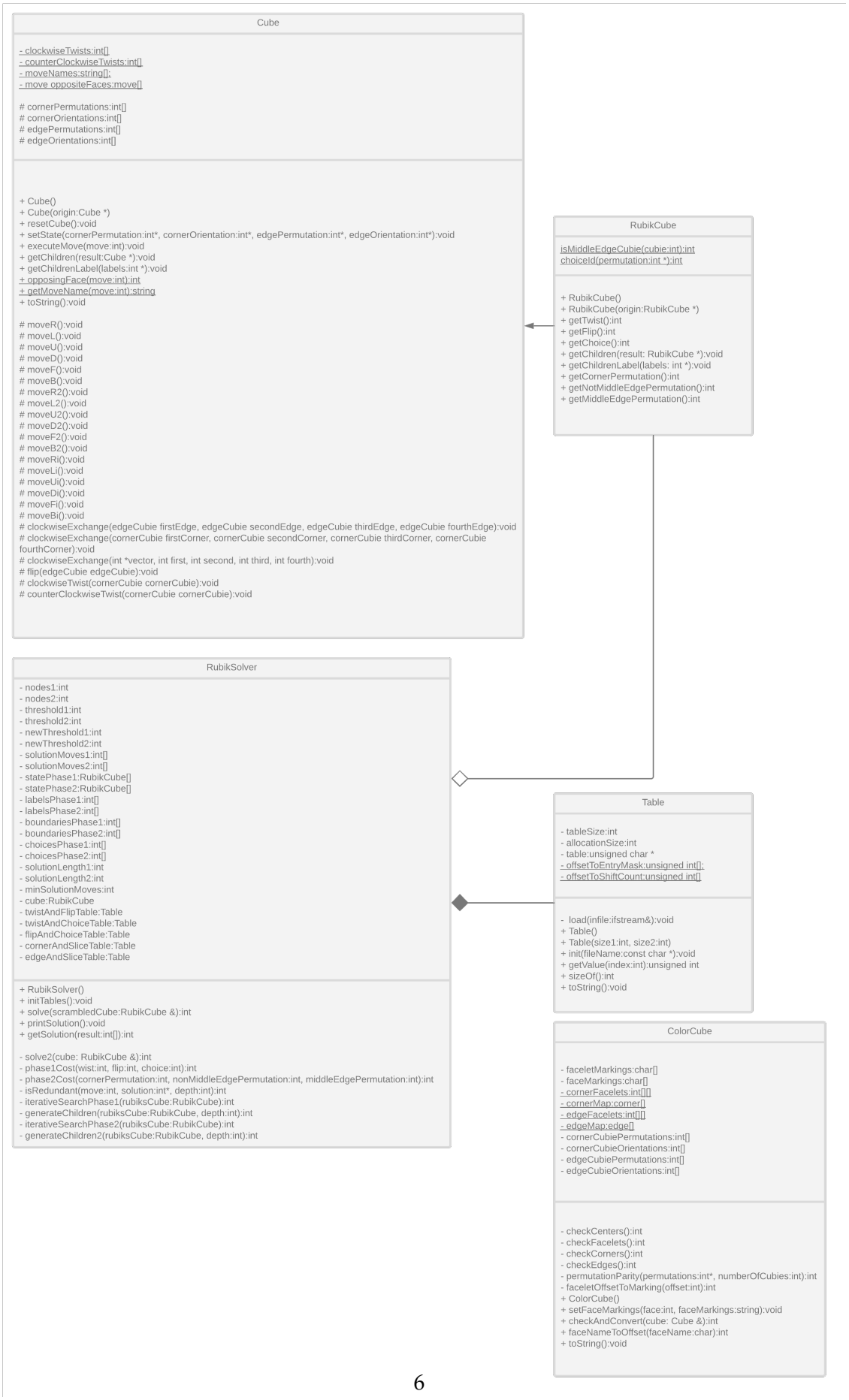

Figure 3: Class Diagram

2.0.3 *Design Reuse*

The software is very extensible: the portion related to the definition of a state and the part in which the search algorithm is implemented are decoupled. The only common part is the heuristic definition. This means that is very easy implement other search algorithms with the same states or new state definitions with the same search algorithms. The source code is available at https://github.com/massimoPerini/XOHW18 and is fully commented.