Edgar Fournival M2 COMASIC

Project report: Internet of Things

Introduction

The goal of this project is to implement a simple measuring device which will send sensors value to a server and set its RGB LED color accordingly.

It is not really useful but the architecture answer the project guidelines. In the future, my project has the great potential to become a $1.50 gadget sold on AliExpress (with free shipping).

In order to do something interesting, I have used an uncommon language : Go. As almost nothing high-level has already been developed, I had to do the port from C/Python (based on SunFounder GitHub examples) to Go. This helped me understand some of the electronics behind the code and it also improved the available embedded code base for this language.

Also, please note that Go is not designed for low-level programming as it has a garbage collector. However, it is becoming more and more popular for embedded development due to its high-level nature and its powerful (cross-)compiler.

1 Setup

1.1 Networking

During the development and the testing of the project, I used an Ethernet cable with a static networking configuration.

The Raspberry Pi, my home computer and the Internet connection were all connected to a Dlink switch.

In order to define a non-volatile network configuration on the Raspberry Pi, the file `/etc/network/interfaces` was edited by adding the following instructions:

```
iface eth0 inet static
  address 192.168.1.11
  netmask 255.255.255.0
  gateway 192.168.1.1
```

My Internet connection is backed by an Orange Livebox which is on a 192.168.1.0/24 network. It supports DHCP but I preferred to give the Raspberry Pi a static configuration in order to ease the configuration of the SSH link between the devices.

1.2 SSH

SSH is a secure transport protocol designed to replace Telnet. It can be used to send commands to a remote device or setup things such as port forwarding, encrypted tunnels, etc. I have decided to setup SSH on the Raspberry Pi in order to drop the need to use the QWERTY keyboard, the mouse and another screen.

The SSH server service has been enabled on the Raspberry Pi, meaning it will start during the boot. This has been achieved with this command:

```
sudo systemctl enable ssh
```

After that, I created a new SSH key on my home computer without a passphrase using `ssh-keygen` with its default options. The key was then sent to the Raspberry Pi using:

```
ssh-copy-id -i ~/.ssh/rpi pi@192.168.1.11
```

To make things easier, I also defined a SSH alias in `~/.ssh/config`:

```
Host rpi
  IdentityFile ~/.ssh/rpi
  User pi
  HostName 192.168.1.11
```

This allowed me to connect to the Raspberry Pi under the alias `rpi` and without providing a password. From now I can type directly `ssh rpi` or use `rpi` in the `rsync` or `scp` commands.

1.3 Hardware

1.3.1 The basics

1. Open the Raspberry Pi in order to access the GPIO pins.
2. Plug the GPIO "rainbow pride" cable to the Raspberry Pi.
3. Plug the GPIO extension board to the other end of the GPIO cable.
4. Plug the GPIO extension board into the middle of the breadboard.
5. Plug a red cable between the `3V3` pin and the `+` column on the left.
6. Plug a black cable between the `GND` pin and the `-` column on the left.

1.3.2 RGB LED module

1. Plug a cable between the `VCC` pin and the `3V3 +` column on the left.
2. Plug the `R` pin into the `GPIO17` pin of the GPIO extension board.
3. Plug the `G` pin into the `GPIO18` pin of the GPIO extension board.
4. Plug the `B` pin into the `GPIO27` pin of the GPIO extension board.

1.3.3 PCF8591 module

1. Plug a cable between the `VCC` pin and the `3V3 +` column on the left.
2. Plug a cable between the `GND` pin and the `3V3 -` column on the left.
3. Plug the `SDA` pin into the `SDA1` pin of the GPIO extension board.
4. Plug the `SCL` pin into the `SCL1` pin of the GPIO extension board.
5. Plug a 4-pins female cable into the `AIN0`, `AIN1`, `AIN2` and `AIN3` of the module.
6. Plug the other end of the cable into unused lines on the breadboard.

1.3.4 Thermistor

1. Plug a cable between the `VCC` pin and the `3V3 +` column on the left.
2. Plug a cable between the `GND` pin and the `3V3 -` column on the left.
3. Plug a cable between the `SIG` pin and `AIN0` from the PCF8591 on the breadboard.

1.3.5 Photoresistor

1. Plug a cable between the `VCC` pin and the `3V3 +` column on the left.
2. Plug a cable between the `GND` pin and the `3V3 -` column on the left.
3. Plug a cable between the `SIG` pin and `AIN1` from the PCF8591 on the breadboard.

1.3.6 Humiture sensor

1. Plug the `VCC` pin into the `5V0` pin of the GPIO extension board.
2. Plug the `GND` pin into the `GND` pin of the GPIO extension board.
3. Plug the `SIG` pin into the `GPIO22` pin of the GPIO extension board.

1.4 Software

1.4.1 The Go toolchain

Then, we will need the Go toolchain for local testing. The package available in Raspbian is too old, so here is the command for the last release:

```
wget https://dl.google.com/go/go1.10.linux-armv6l.tar.gz
sudo tar -C /usr/local -xzf go1.10.linux-armv6l.tar.gz
echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.bashrc
```

After that, we need to setup Go environment in a specific folder and permanently export the `GOPATH` environment variable:

```
sudo su -
mkdir -p ~/go
echo "export GOPATH=~/go" >> ~/.bashrc
source ~/.bashrc
```

Note we are doing this with the `root` user because only the superuser can play with PWM (used by the RGB LED).

1.4.2 Cloning the repository

Git is already installed on the Raspberry Pi. We will use it to clone the repository of my project hosted on GitHub.

Note that the repository must be cloned in `$GOPATH` in order to be able to use `go get`:

```
sudo su -
export IOTDIR=$GOPATH/src/github.com/efournival
mkdir -p $IOTDIR && cd $IOTDIR
git clone https://github.com/efournival/iot-project.git
cd iot-project
```

1.4.3 Enabling the I2C bus

The I2C bus is used by the PCF8591 analog-to-digital converter in order to read the temperature from the thermistor.

First, connect to the device using SSH (for example, `ssh rpi`) and run the configuration utility:

```
sudo raspi-config
```

Then:

1. Enter `9 Advanced Options`.
2. Enter `A6 I2C`.
3. Answer “yes” to the question “Would you like the ARM I2C interface to be enabled?”.
4. Wait for the confirmation then quit with the escape key.

The device driver should now be available under `/dev`, you can quickly check by running `ls /dev/i2c-1`.

2 Architecture

My project has a simple architecture:

- The Raspberry Pi is collecting sensor data and sending it to the server.
- The server computes the LED color from sensor data before sending it back to the Raspberry Pi.
- The Raspberry Pi receives the color and sets it accordingly on the hardware.

UDP has been chosen for the networking part as we do not care if we lose a packet. Also, it is more flexible than TCP, which is a stream protocol and requires connection.

3 Development

3.1 Porting the code

The Raspberry Pi has a great existing ecosystem with a big available code base in C and Python. Almost nothing do exist in Go. The challenge of this project, that I modified a little in order to do something original, was to do the same as my classmates with the low-level part.

Each chosen module was tricky to implement except the thermistor and the photoresistor, because only the PCF8591 was hard to port to Go. As they are both connected to it, the code was quite easy to do.

Starting with the RGB LED, it uses PWM pins which were not supported by the GPIO library I initially chose. I finally used PeriphIO[1] which is a project backed by Google to promote embedded programming with Go. My code for this LED seems to be working but neither the SunFounder's original C code nor mine's succeeded to display right colors. I think there is a problem with my RGB LED module but I can still display red, blue, green or yellow colors. Intermediate ones are either light blue or almost white.

The PCF8591 code was also difficult to port as the implementations in C or Python were too much relying on the specifics libraries (WiringPi for example). I even needed the original data sheet[2] from the manufacturer in order to port it successfully. Once this was done, porting the thermistor and the photoresistor code has been quick to achieve.

Finally, the DHT11 (humiture) sensor is a piece of unreliable electronics and keeps failing to send the sensor data 30-50% of the time. The Raspberry Pi implementation with WiringPi was so bad that I needed to port an Arduino code and tweak it in order to improve reliability. I finally achieved less failures with my Go implementation than with the SunFounder's C code.

3.2 Client/server

Go is great language for everything related to networking. I used the UDP standard library and added more high-level wrappers around it in order to have readable final device and server codes.

My package in the folder *udp* contains the wrappers.

3.3 Threads, synchronization, etc.

All of these was not necessary due to the asynchronous nature of the Go language.

I have used the `go` keyword to start concurrent functions like UDP socket listening and smooth RGB LED color changing.

Also, client and server codes use a callback when receiving an UDP packet.

[1]`https://periph.io/`

[2]`https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf`

4 Testing

On the device, change the current directory to the package *device*. Then run `go get` in order to automatically retrieve the dependencies (PeriphIO). Then run `go build` and `./device HOST_IP`. `HOST_IP` must be the IP of the host on the same network (most probably Wi-Fi).

Client and server can be launched in any order since there is no connection.

On the host, change the directory to *server*. Do not forget to clone the repository in `$GOPATH` exactly as we did with the device, but without becoming the superuser first. Run `go get`, `go build` and `./server`.

Finally, watch how the RGB LED color is changing as the sensors data is changing too. Impressive.