DFRobot project



Contents

Introduction 3

Raspberry Pi configuration 4

Backing up the SD card image of the Raspberry Pi 4

Mounting the Raspberry Pi SD card in Ubuntu 4

Connecting the Raspberry Pi to the MacBook via the ethernet cable 5

Network configuration in /etc/network/interfaces 5

Wifi configuration in /etc/wpa\_supplicant/wpa\_supplicant.conf 5

Connecting the Raspberry Pi to the wireless network (no Ethernet cable) 6

Logging into the Raspberry Pi with SSH and with VNC 7

Logging into the Raspberry Pi with X11 forwarding with SSH 7

Raspberry Pi file sharing 7

Raspberry Pi camera module, MJPEG streaming 7

Local record of video stream in H264 8

Capturing the http MJPEG stream while driving 8

Uploading files from the Raspberry to Google Drive 9

Apache webserver port configuration 9

Apache webserver password protection 10

Enable .py scripts with CGI in Apache2 10

Bash scripts versus Python scripts with CGI in Apache2 10

Enabling I2C on the Raspberry Pi 11

Enabling the hardware watchdog on the Raspberry Pi 11

Creating a custom software watchdog bash script ‘own\_watchdog.sh’ on the Raspberry Pi to check the network connection 12

Scheduling a task to run at regular times 13

Threading in Python 14

OpenCV attempt 1: building and installing OpenCV from source on the Raspberry Pi 14

OpenCV attempt 2: installing a precompiled OpenCV on the Raspberry Pi 15

OpenCV attempt 3 (and final): installing the OpenCV libraries and python-opencv 15

OpenCV pattern matching 15

OpenCV motion detection 16

Arduino Uno configuration 17

Measurements 18

Operate the DFRobot using only 3G or 4G 21

Circuit diagram 23

# Introduction

This project was started beginning 2015 to get familiar and (even more) enthousiastic about embedded systems, embedded Linux, Robotics. Its fits very well the curriculum of the Fontys FHICT Technology study. Topics addressed are:

* Embedded systems
* Embedded Linux
* Networking
* Web server, webpage development
* Video streaming over internet
* ROS
* Programming C, C++, Python, Bash

To get started the following hardware is used:

* DFRobot 4WD Arduino Mobile Platform
* Raspberry Pi Model B+ Computer Board, replaced by Raspberry Pi 2B on 24/07/2015.
* Raspberry Pi Camera Module
* DFRduino Uno USB Microcontroller V2.0
* DFRobot Arduino Compatible Motor Shield (2A)
* Lynxmotion Pan and Tilt Kit / Aluminium
* Netgear Wireless N Adapter 150 Mbps (USB) (WNA1100-100PES)
* GY-271 module with HMC5883L 3-Axis Digital Compass
* 4x HC-SR04 ultrasonic sensor

The project will never be ‘finished’ and is perfect for keeping up to date with the latest technologies. At the moment the following phases are foreseen:

**Phase one: feasibility**

In this phase I want to investigate the areas which I see as high risk and which are crucial for success of the project. Better to encounter problems in these areas right away then later in the project.

* Spike to see if it was possible to have low latency (< 2 sec) streaming video from the Raspberry Pi to a standard web browser (no plugins). This is not trivial and a lot of discussions can be found in forums.
* Spike to see how to control the Pi Camera and the Pi I/O pins from a standard web browser. Different techniques can be used and it would be good to use the right one for this project.

**Phase two: Basic system up and running**

* Robot platform controlled from a standard web browser containing streaming video. Status information will also be available.
* Next to that a charging station will be built and it should be possible to manually drive to the driving station and attach.

**Phase three: Adding more features / intelligence**

In this phase I want to add intelligence and features.

* Adding sensors like for measuring distance, temperature etc…
* Let the robot drive independently every hour and upload the video to Google Drive.
* Let the robot detect motion and upload the video to Google Drive.
* Let the robot automatically find and attach to its charging station when needed.
* Adding servo’s for grabbing.
* Let the robot create a map of the environment so that it can orientate itself and one can give commands like: go to room X position Y. This is where ROS comes in.
* Automatic uploading of video and log files to Google Drive.

# Raspberry Pi configuration

## Backing up the SD card image of the Raspberry Pi

To create an image of the Pi SD card the dd command can be used. Put the SD card in the Mac and identify the 8 GB SD card disk with ‘diskutil list’. The Identifier will be something like ‘disk1’. Then in the dd command one can use /dev/disk1 for the device or for higher speed C) raw disk /dev/rdisk1 which is ok when using the dd command.

sudo dd if=/dev/rdisk1 of=/Users/fhict/ReneB/GitHub/rbakx/DFRobot\_SDcardBackup\_TooLargeForGithub/SDcardBackup20150124.img bs=1m

or to save space:

sudo dd if=/dev/rdisk1 bs=1m | gzip > /Users/fhict/ReneB/GitHub/rbakx/DFRobot\_SDcardBackup\_TooLargeForGithub/SDcardBackup20150124.gz

This will create an image you can use to create a new SD card. This image can be restored on an SD card using again the dd command:  
sudo dd bs=1m if=/Users/fhict/ReneB/GitHub/rbakx/DFRobot\_SDcardBackup\_TooLargeForGithub/SDcardBackup20150124.img of=/dev/rdisk1

To allow overwriting the disk you have first to unmount /dev/disk1 with: sudo diskutil unmountDisk /dev/disk1 otherwise the resource will be busy. Note that unmounting a disk is not the same as ejecting it from the file explorer.  
When using an 8 GB SD card it might be needed to expand the root file system to make sure the complete memory card is used. This can be done using ‘sudo raspi-config’ and selecting the ‘Expand Filesystem’ option. The data will remain.

## Mounting the Raspberry Pi SD card in Ubuntu

When the Raspberry Pi SD card is corrupt and connection to the DFRobot is not possible anymore, it might be possible to fix it by changing a file on the SD card. Then it is useful to be able to write Raspbian files when inserting the SD card in the Mac. Because the Raspbian Linux partition cannot be read normally under OS X (except with Fuse for OS X), we make use of Ubuntu and WMware Fusion (VirtualBox did not seem to work here).  
On the Mac, start Ubuntu in VMware Fusion. Insert the SD card in the Mac and eject it under OS X. In VMware Fusion: Ubuntu 64 bit -> Virtual Machine -> USB & Bluetooth -> Connect Apple Internal Memory Card Reader. In Ubunto the memory card should appear. In the file browser it is read only so we need to mount it to be able to acces it in terminal. Open terminal and list the devices with ‘sudo fdisk –l’. There the name of the Linux partition of the SD card can be seen, e.g. ‘/dev/sdb2’. This device can be mounted with ‘sudo mkdir /sdcard’ and then ‘sudo mount /dev/sdb2 /sdcard’. Then the Linux partition can be accessed at ‘/sdcard’ from the terminal. ‘sudo’ can be used to write files if needed. Note that some files cannot be seen as standard user as they are readable only by the root.

## Connecting the Raspberry Pi to the MacBook via the ethernet cable

We want to connect the Pi to the MacBook through the Ethernet cable so we can control the Pi with the keyboard / screen of the MacBook and also provide internet access to the Pi. The latter is done by enabling internet sharing on the MacBook.

* First configure the MacBook Ethernet network settings to static IP 192.168.2.1. It must be this address because this is the fixed IP address which the OS X Mavericks uses when enabling internet sharing. The address 192.168.2.1 address means that the Pi must have a static address in the same subnet, e.g. 192.168.2.2. This is accomplished by setting this in **/etc/network/interfaces**. Also the gateway of the Pi has to be set there and this must be the IP of the MacBook: IP 192.168.2.1.
* Internet sharing from MacBook:  
  When enabling internet sharing on the MacBook the MacBook apparently always uses 192.168.2.1. This is on a different subnet then the Wifi, so the MacBook acts as a router.
* A reboot might be needed before internet sharing is really working. Still the setup seems not very reliable.

## Network configuration in /etc/network/interfaces

For the wlan0 interface (Wifi) DHCP is used. Using DHCP is convenient so the Pi also can operate in other networks, like in a phone’s tethering network! A fixed DHCP address can be supplied by the router, which is convenient so the Pi always has the same IP address which is also needed for port forwarding. This means that the router must assign a static DHCP IP address (192.168.1.42) to the Pi (using its MAC address).  
Below note that for the ethernet interface no DHCP is used but static IP address settings which is convenient because the Ethernet DHCP service on the Mac apparently cannot supply a static DHCP IP address, so finding out the Pi’s IP address is difficult then.

auto lo

iface lo inet loopback

iface eth0 inet static

address 192.168.2.2

netmask 255.255.255.0

gateway 192.168.2.1

allow-hotplug wlan0

iface wlan0 inet manual

wpa-roam /etc/wpa\_supplicant/wpa\_supplicant.conf

iface default inet dhcp

## Wifi configuration in /etc/wpa\_supplicant/wpa\_supplicant.conf

Below note that two access points are specified to allow roaming. Also note the network section for the tethering network of a phone!

ctrl\_interface=DIR=/var/run/wpa\_supplicant GROUP=netdev

update\_config=1

network={

ssid="wifiwifiwifi2"

psk="fill in password here"

# Protocol type can be: RSN (for WP2) and WPA (for WPA1)

proto=RSN

# Key management type can be: WPA-PSK or WPA-EAP (Pre-Shared or Enterprise)

key\_mgmt=WPA-PSK

# Pairwise can be CCMP or TKIP (for WPA2 or WPA1)

pairwise=CCMP

#Authorization option should be OPEN for both WPA1/WPA2 (in less commonly used $

auth\_alg=OPEN

}

network={

ssid="wifiwifiwifi"

psk="fill in password here"

# Protocol type can be: RSN (for WP2) and WPA (for WPA1)

proto=RSN

# Key management type can be: WPA-PSK or WPA-EAP (Pre-Shared or Enterprise)

key\_mgmt=WPA-PSK

# Pairwise can be CCMP or TKIP (for WPA2 or WPA1)

pairwise=CCMP

#Authorization option should be OPEN for both WPA1/WPA2 (in less commonly used $

auth\_alg=OPEN

}

network={

ssid="AndroidAP"

psk="AndroidAP"

# Protocol type can be: RSN (for WP2) and WPA (for WPA1)

proto=RSN

# Key management type can be: WPA-PSK or WPA-EAP (Pre-Shared or Enterprise)

key\_mgmt=WPA-PSK

# Pairwise can be CCMP or TKIP (for WPA2 or WPA1)

pairwise=CCMP

#Authorization option should be OPEN for both WPA1/WPA2 (in less commonly used $

auth\_alg=OPEN

}

Testing internet speed from command line:  
wget -O /dev/null http://speedtest.wdc01.softlayer.com/downloads/test10.zip

## Connecting the Raspberry Pi to the wireless network (no Ethernet cable)

* Configure the file /etc/network/interfaces and the /etc/wpa\_supplicant/wpa\_supplicant.conf like above.
* The Pi will get an IP address assigned by DHCP: 192.168.1.42.

## Logging into the Raspberry Pi with SSH and with VNC

* Connect to Pi via SSH: ssh [pi@192.168.1.42](mailto:pi@192.168.1.42), password ‘raspberry’.
* Install VNC server on Pi and VNC client on MacBook.
* Start up VNC server on Pi with: ‘vncserver :1’.
* Start up VNC client on MacBook and connect to 192.168.1.42, password ‘raspberry’.

## Logging into the Raspberry Pi with X11 forwarding with SSH

* Connect to Pi via SSH: ssh –Y [pi@192.168.1.42](mailto:pi@192.168.1.42), password ‘raspberry’. Note the ‘-Y’ which enables trusted X11 forwarding. ‘–X’ can also be used but then it will timeout after the default 20 minutes ‘ForwardX11Timeout’ timeout of untrusted connections.
* Install XQuartz on MacBook and set X11 output preferences to ‘Full-screen mode’.
* Start ‘/etc/X11/Xsession’ on the Pi. This will automatically launch XQuartz on the Mac showing the X11 window.
* With cmd-option A one can go back from full screen Pi to the Mac.

## Raspberry Pi file sharing

At http://4dc5.com/2012/06/12/setting-up-vnc-on-raspberry-pi-for-mac-access/ it is described how to set up file sharing. Netatalk is installed which is an open source AFP (Apple Filing Protocol) file server. Then Avahi is installed which is a zero configuration service discovery protocol. The file ‘/etc/avahi/services/afpd.service’ is created with the avahi settings (a.o. TCP/IP port 548 is specified). Avahi is then started as a deamon which will start automatically after each reboot.

In /etc/netatalk/AppleVolumes.default at the end there is:

~/ "Home Directory"

which is the default folder to share. There you can add an extra path to share like:

/etc "/etc"

/lib “/lib”

Then on the Mac, in finder, press cmd-K (connect to server) and enter the Pi IP address and password. Then the Pi shared folder shows up in the shared section in the Finder sidebar.  
When writing to a Pi folder like ‘/etc’ from the Mac the permission is denied. This is because the user ‘pi’ is not the owner or member of the group of that folder (root is) and the permission is on 755. When writing to ‘/etc’ is needed, on the Pi do: ‘sudo chmod 777 /etc’.

After writing put back permission to 755 with ‘sudo chmod 755 /etc’.

## Raspberry Pi camera module, MJPEG streaming

For MJPEG streaming mjpg-streamer is installed following <http://petrkout.com/electronics/low-latency-0-4-s-video-streaming-from-raspberry-pi-mjpeg-streamer-opencv/> which is a mjpg-streamer fork of Liam Jackson. The installation directory chosen is /opt/mjpeg-streamer instead of /usr/src/mjpeg-streamer.  
For MJPEG streaming also the UV4L (User space Video For Linux) framework from Luca Risolia can be used. However, the MJPEG stream generated by this framework has problems when trying to read it with OpenCV (premature ending of jpg frames). The quality of this framework is the same as the quality of mjpeg\_streamer.

To start the camera and MJPEG stream:

LD\_LIBRARY\_PATH=/opt/mjpg-streamer/mjpg-streamer-experimental/ /opt/mjpg-streamer/mjpg-streamer-experimental/mjpg\_streamer -i "input\_raspicam.so -vf -hf -fps 10 -q 10 -x 800 -y 600" -o "output\_http.so -p 44445 -w /opt/mjpg-streamer/mjpg-streamer-experimental/www" > /dev/null 2>&1.

The stream can be viewed via the MJPG-Streamer web page at 192.168.1.42:44445 or directly at 192.168.1.42:44445/stream\_simple.html.

To stop the stream use:

‘killall mjpg\_streamer > /dev/null 2>&1’

Stopping the stream can also be done by installing uv4l-raspicam-extras and using ‘/usr/sbin/service uv4l\_raspicam stop’ but this seems not to be reliable.

The stdout generated by the Bash CGI script is sent back to the browser, to capture all stdout output which must not be returned to the html.

This can be done using '> /dev/null 2>&1' or writing to a logfile.

## Local record of video stream in H264

Recording a H264 video stream can be done using the ‘raspivid’ command. For example:

‘raspivid -o /home/pi/DFRobotUploads/dfrobot\_pivid.h264 -w 1280 -h 720 -vf -hf -t 60000 > /dev/null 2>&1 &’ to record a one minute 1280x720 video with a vertical and horizontally flipped image. This is raw H264 video which cannot be played by most media players. To enable the ‘www-data’ user to run this command the ‘DFRobotUploads’ folder must have ‘777’ permission as the folder where the log files are written.

To wrap it into a MP4 container format install gpac with ‘sudo apt-get install -y gpac’. After this the video can be converted with: ‘MP4Box -fps 30 -new -add /home/pi/DFRobotUploads/dfrobot\_pivid.h264 /home/pi/DFRobotUploads/dfrobot\_pivid.mp4 > /dev/null 2>&1’.

raspivid can be stopped using ‘killall raspivid’.

To enable different users like ‘pi’ and ‘www-data’ to both recreate these video files the permission has to be changed to ‘666’ once.

Recording the local video stream with ‘raspivid’ cannot be done together with streaming the MJPEG video with UV4L.

## Capturing the http MJPEG stream while driving

Capturing the http MJPEG stream and convert it to a viewable format can be done by using VLC. At the moment VLC on the Raspberry does not seem to be able to transcode, but capturing and encapsulating it in ‘MPEG TS’ format works.

This can be done with:

‘cvlc http://localhost:44445/stream\_simple.html --sout '#standard{mux=ts,dst=/home/pi/DFRobotUploads/dfrobot\_pivid\_man.mp4,access=file}' --run-time=60 vlc://quit > /dev/null 2>&1 &’.

The ‘--run-time=60 vlc://quit’ part is to let VLC stop recording after 60 seconds, in case the driver forgets to stop it when finished.

This format is not recognized by Google Drive but it can be played by VLC.

As soon as transcoding works use:

‘#transcode{vcodec=h264}:standard{mux=ts,dst=/home/pi/DFRobotUploads/dfrobot\_pivid\_man.mp4,access=file}' --run-time=60 vlc://quit > /dev/null 2>&1 &’.

This format is recognized by Google Drive.

The capturing can be stopped using ‘killall vlc > /dev/null 2>&1’.

Just capturing and storing the mjpeg stream to a file can be done using curl:

‘curl -s -o /home/pi/DFRobotUploads/dfrobot\_pivid\_man.mjpeg http://localhost:44445/stream\_simple.html > /dev/null 2>&1 &’.  
However this file cannot be played by a standard media player.

## Uploading files from the Raspberry to Google Drive

To upload files from the Raspberry to Google Drive ‘gdrive’ is used, see <http://www.webupd8.org/2014/09/gdrive-simple-google-drive-cli-client.html> and <https://github.com/prasmussen/gdrive> for the github the repository.

From bash script a file can be uploaded too Google Drive using:

‘sudo -u www-data drive upload -p 0B1WIoyfCgifmMUwwcXNqeDl6U1k -f /home/pi/DFRobotUploads/dfrobot\_pivid.mp4 > /dev/null 2>&1 &’.

Explanation of this command:

When ‘drive’ runs as a new user and tries to connect to Google Drive, Google will ask for a verification code. It provides a link to the Google Drive login page to get a verification code. One has to login with a valid Google Drive account to get a verification code for this account. The verification code has to be provided once and will be stored in the ‘.gdrive’ subfolder. From then on the new user has access to the Google Drive account. This means that also the verification code for the ‘www-data’ user must be stored as ‘www-data’ is the user used by the Apache webserver. This is done by first running the ‘drive’ command with ‘sudo –u www-data’ and then providing the verification code which is then stored. However, when the running the ‘drive’ command from the ‘dfrobot.sh’ bash script which is called by the Apache server (as user www-data), again a verification code is asked. Apparently the user is still different in a way. To solve this the ‘drive’ command is also run as ‘sudo –u www-data’ from the dfrobot.sh bash script. Therefore the www-data user must be member of the sudo group which can be done with ‘sudo adduser www-data sudo’ .

To upload into the 'DFRobotUploads' folder, the -p option is used with the id of this folder. This id can be obtained using 'drive list -t DFRobotUploads'. When the 'DFRobotUploads' folder is changed, a new id has to be provided. The uploaded files have a distinctive name to enable finding and removing them again with the 'drive' utility.

## Apache webserver port configuration

To set the correct Apache webserver port edit ‘/etc/apache2/ports.conf’ and change port 80 to 44444:

NameVirtualHost \*:44444

Listen 44444

In addition in ‘/etc/apache2/sites-enabled/000-default’ change the first line from <VirtualHost \*:80> to <VirtualHost \*:44444>.

## Apache webserver password protection

See <http://www.debiantutorials.com/password-protecting-a-directory-with-apache-and-htaccess/>.

The DFRobot website is protected by letting the Apache server protect the ‘/var/www’ directory, where the index.html file is located. The file ‘/etc/apache2/sites-enabled/000-default’ is modified to contain:

<Directory /var/www/>

AllowOverride All

</Directory>

This enables ‘.htaccess’ file support in ‘/var/www’.  
A ‘.htaccess’ file is created and placed in /var/www (where the index.html file is located) to protect the complete DFRobot website:

AuthType Basic

AuthUserFile /etc/apache2/.htpasswd

AuthName "Enter password"

Require valid-user

It points to ‘/etc/apache2/.htpasswd’ which contains the password for user ‘reneb’. The command ‘htpasswd -c /etc/apache2/.htpasswd reneb’ can be used to generate a new password for user ‘reneb’.

## Enable .py scripts with CGI in Apache2

Open ‘/etc/apache2/sites-enabled/000-default’ (this is a link to the file ‘default’) and add  
’AddHandler cgi-script .py’ to the section

<Directory "/usr/lib/cgi-bin">  
...  
</Directory>

Then reload Apache’s configuration using ‘sudo service apache2 reload’.

When running Python scripts from Apache2 through CGI the user is ‘www-data’. So scripts and external program run must have the proper group permissions.  
When using the camera module the VideoCore device ‘/dev/vchiq’ is used. The default permissions are:  
crw-rw---T 1 root video 250, 0 Jan 1 1970 /dev/vchiq  
These permissions mean that www-data cannot read from ‘/dev/vchiq’ which will give the ‘failed to open vchiq instance’ error when trying to use the camera from CGI Python. It can be seen that the user group ‘video’ is assigned to ‘/dev/vchiq’.

To be able to run the camera from CGI Python: ‘sudo chmod 666 /dev/vchiq’ works, but the original permissions are reset on reboot.  
So a permanent solution is to make the www-data (www-data is the user under which the Apache2 web server runs) member of the video group. This can be done with: ‘sudo adduser www-data video’.

## Bash scripts versus Python scripts with CGI in Apache2

When using Python CGI the problem is encountered that for each http POST method call the Python interpreter on the Pi must be loaded and the Python script executed. This can take 1 to 3 seconds, depending if video is streaming or not. As this is not an acceptable delay I decided to switch from Python CGI to bash CGI which is much faster.

The bash scripts used can be called without ‘bash’ in front , but make sure it start with ‘#!/bin/bash’ (and not #!/bin/sh) to use the bash shell.

## Enabling I2C on the Raspberry Pi

Information taken from <http://www.legomindstormsrobots.com/arduino/connecting-arduino-raspberry-pi-i2c/>

Enable the I2C hardware by adding at the end of /etc/modules:  
i2c-bcm2708

i2c-dev

Next install the i2c-tools utility:

‘sudo apt-get install python-smbus’.

‘sudo apt-get install i2c-tools’.

If /etc/modprobe.d/raspi-blacklist.conf exists, comment out the lines  
blacklist spi-bcm2708

blacklist i2c-bcm2708

From kernel version 3.18 on (‘uname –a’ shows kernel version): to use GPIO2 (pin 3) and GPIO3 (pin 5) as SDA and SCL respectively, add ‘dtparam=i2c1=on’ to the end of /boot/config.txt.  
Add the pi user to the i2c group, to do this type 'sudo adduser pi i2c'.

The I2C devicescan be scanned using ‘i2cdetect -y 1’.

To allow usage of the I2C device by www-data:  
‘sudo adduser www-data i2c'.

Finally, in ‘/lib/udev/rules.d/60-i2c-tools.rules’, change  
KERNEL=="i2c-[0-9]\*", GROUP="i2c", MODE="0660"

to  
KERNEL=="i2c-[0-9]\*", GROUP="i2c", MODE="0666"

to have the I2C device readable by the i2c group.  
Then reboot using ‘sudo reboot’.

## Enabling the hardware watchdog on the Raspberry Pi

To make the system more robust a hardware watchdog is used. The idea is to reboot the system as soon network connection is lost.

* First install the watchdog kernel module with:  
  ‘modprobe bcm2708\_wdog’
* To make the watchdog module loaded automatically at boot time, set in ‘/etc/default/watchdog’:  
  # Start watchdog at boot time? 0 or 1  
  run\_watchdog=1  
  # Load module before starting watchdog  
  watchdog\_module="bcm2708\_wdog"  
  # Specify additional watchdog options here (see manpage).
* Install a watchdog daemon and the chkconfig tool (to activate and deactivate services) with ‘sudo apt-get install watchdog chkconfig’.
* Enable the watchdog service with ‘sudo chkconfig watchdog on’.
* Start the watchdog with ‘sudo /etc/init.d/watchdog start’.
* Edit ‘/etc/watchdog.conf’ to set what should be checked by the watchdog daemon. For example to check if the router can be pinged:  
  ‘ping = 192.168.1.254’  
  To check if a file has been updated the ‘file’ and ‘change’ settings can be used. Unfortunately both options do not work, the Raspberry Pi is rebooting continuously. This is a known problem. It seems that when the watchdog daemon is not started correctly when using these options. It seems to do with:  
  *For those who want the gruesome details: wd\_keepalive is supposed to prevent the watchdog (hardware) from timing out during boot. After boot-up watchdog (software) is meant to kill wd\_keepalive and take over. Problems with the startup scripts mean that this isn't happening: wd\_keepalive continues to run and conflicts with watchdog (software), causing the watchdog (hardware) to time-out prematurely. In fact wd\_keepalive isn't needed, because the watchdog (hardware) isn't enabled until watchdog (software) loads. There, clear as mud.*Also at instructables found the remark: ‘If the Raspberry Pi takes longer to bootup than 14 seconds (or to whatever value you set Wto), the WatchDog can fire which puts the Raspberry Pi in an infinite bootup sequence’. The 14 seconds is a hardware limit so it cannot be increased.
* Manually starting the watchdog daemon with these settings after boottime does work, but is not practical.  
  Therefore we currently use the setting:  
  #file = /usr/local/bin/own\_watchdog.sh  
  #change = 40  
  watchdog-device = /dev/watchdog  
  realtime = yes  
  priority = 1  
  The ‘own\_watchdog.sh’ bash script (see below) runs continuously and checks for the network connection. If all is ok it updates its modification timestamp to let the watchdog know all is ok. If the timestamp is not updated for 40 seconds, the system will reboot.  
  As mentioned above the ‘change option does not work, the Raspberry Pi gets in a infinite reboot loop.   
  Therefore the ‘file’ and ‘change’ lines are commented out. What does work is that the watchdog is running and when for example the Wifi dongle is unplugged and plugged in again, the system reboots and thus recovers. In combination with the ‘own\_watchdog.sh script which reboots when ping fails, we have a robust failure recovery.
* The watchdog can be tested by adding watchdog\_options="-v -q" to /etc/default/watchdog. With these options the watchdog will only log but not actually reboot. The log can be checked in /var/log/daemon.log.
* Update 25072015: The Raspberry Pi 2B seems to have the same problem as described above. Also at instructables found the remark: ‘If the Raspberry Pi takes longer to bootup than 14 seconds (or to whatever value you set Wto), the WatchDog can fire which puts the Raspberry Pi in an infinite bootup sequence’.

## Creating a custom software watchdog bash script ‘own\_watchdog.sh’ on the Raspberry Pi to check the network connection

A custom watchdog in the form of a bash script is created. This script checks whether the router (192.168.1.254) or the Samsung Wifi hotspot (192.168.43.1, this is hardcoded in Android) can be pinged. If both cannot be pinged, it does not update its modification timestamp. This will be detected by the watchdog which will then reboot the system in about 15 seconds.  
  
#!/bin/bash

# Start with a touch of this script to feed the watchdog daemon a.s.a.p.

prompt=$(basename $0)

# Reset the log file to zero length if the size gets too large.

if [ $(stat -c %s /home/pi/log/dfrobot\_log.txt) -gt 1000000 ]

then

echo -e "\*\*\*\*\* $(date), $prompt: START LOG \*\*\*\*\*" > /home/pi/log/dfrobot\_log.txt

else

echo -e "\n\*\*\*\*\* $(date), $prompt: START LOG \*\*\*\*\*" >> /home/pi/log/dfrobot\_log.txt

fi

count=0

touch -m $0

while [ 1 ]

do

# Start with a sleep to give network time to come up.

echo going to sleep

sleep 10

# Check for router and Android Wifi hotspot

if [ "$(ping -c 1 -W 2 192.168.1.254 | grep '100% packet loss')" -a "$(ping -c 1 -W 2 192.168.43.1 | grep '100% packet loss')" ]

then

echo "\*\*\*\*\* $(date), $prompt: ping failed, going to restart network!" >> /home/pi/log/dfrobot\_log.txt

echo ping not ok, going to restart network

sudo ifconfig wlan0 down && sudo ifconfig wlan0 up

sudo /etc/init.d/networking restart

sudo dhclient wlan0

else

# Touch this script to feed the watchdog daemon.

touch -m $0

# Write to logfile every 5 minutes when ping is ok.

if [ $count -ge 30 ]

then

echo "\*\*\*\*\* $(date), $prompt: ping ok" >> /home/pi/log/dfrobot\_log.txt

count=0

fi

fi

count=$((count+1))

done  
  
To have this script called after reboot add to the end of /etc/rc.local:  
‘/usr/local/bin/own\_watchdog.sh &’.

This way we have created:

* A hardware watchdog enabled by ‘run\_watchdog=1’ in ‘/etc/default/watchdog’ which is fed by a watchdog daemon configured in ‘/etc/watchdog.conf’.
* A software watchdog (‘/usr/local/bin/own\_watchdog.sh’)

These watchdogs run independent of each other.

‘/etc/init.d/watchdog status‘ or ‘ps –ef’ can be used to check that the watchdog daemon is running.

## Scheduling a task to run at regular times

To schedule a task at regular times the standard ‘cron’ tool can be used. Use ‘crontab –e’ to edit the cron table.

To run the ‘run\_dfrobot.py’ script every hour add the entry:

0 \* \* \* \* /usr/local/bin/run\_dfrobot.py -nomove –fullrun

To run the ‘run\_dfrobot.py’ script after a reboot:

@reboot /usr/local/bin/run\_dfrobot.py -fullrun &

Or add it at the end of /etc/rc.local:

/usr/local/bin/run\_dfrobot.py -fullrun &

When running from cron the PATH environment variable does not contain ‘/usr/localbin’ so use the full paths for these commands in the run\_dfrobot.py script.

Setting the timezone right can be done with:

‘sudo cp /usr/share/zoneinfo/Europe/Amsterdam /etc/localtime‘.

To check in bash whether there are active connections one can use:

‘if [ $(netstat | grep 44444 | wc -l) -gt 0 ]; then’.

This way connections to port 44444 are counted. This can be handy to prevent the robot from taking autonomic actions when a person is operating it through the webbrowser.

## Threading in Python

For reading the images from the MJPEG input stream a separate thread is created using the Python method thread.start\_new\_thread(). This thread reads the input stream at maximum speed while another thread can do the image processing. This solves the input buffering problem when reading from the MJPEG input stream. Communication between the threads is done using lock objects (not semaphores, which are a more advanced locking mechanism).

## OpenCV attempt 1: building and installing OpenCV from source on the Raspberry Pi

OpenCV (Open Source Computer Vision) is used by the robot to recognize markers in its environment, and use these to navigate. Installing OpenCV is done by following the guide on <http://www.pyimagesearch.com/2015/02/23/install-opencv-and-python-on-your-raspberry-pi-2-and-b/>. However, step 9 – Setup the build failed. It turned out that the default installed gcc 4.6 was corrupt and gave a ‘segmentation fault’ when trying to compile any C++ file. As a resolution gcc 4.7 was installed following <http://www.rpiblog.com/2014/07/installing-gcc-on-raspberry-pi.html>. After this step 9 - Compile OpenCV failed at 75%. The file ‘libpython2.7.so’ seemed to be corrupt. This was solved by issuing ‘sudo apt-get -y remove libpython2.7 python-dev python2.7-dev’ and then ‘sudo apt-get -y install libpython2.7 python-dev python2.7-dev’.  
Then openCV seemed to to be installed from Python but when issuing ‘cv2.imshow("Image", image)’ there is the error: ‘OpenCV Error: Unspecified error (The function is not implemented. Rebuild the library with Windows, GTK+ 2.x or Carbon support. If you are on Ubuntu or Debian, install libgtk2.0-dev and pkg-config, then re-run cmake or configure script) in cvShowImage, file /home/pi/opencv-2.4.10/modules/highgui/src/window.cpp, line 501’. It seems to be related to a missing libgtk. This is a toolkit for creating graphical user interfaces required for the OpenCV highgui, see step 3 of OpenCV installation guide mentioned above. However reinstalling libgtk2.0-dev did not help.

This means that in the end building and installing OpenCV on the Rasberry Pi did not work out.

## OpenCV attempt 2: installing a precompiled OpenCV on the Raspberry Pi

From <https://github.com/Nolaan/libopencv_24/releases> a precompiled version of OpenCV for the Raspberry can be downloaded. It can be installed with ‘dpkg -i libopencv\_2.4.11.deb’. After doing this an ‘import cv2’ sais ‘ImportError: No module named cv2’. This is fixed after installing the Pyhon bindings with: ‘sudo apt-get install libopencv-dev python-opencv’. However then an ‘import cv2’ gives a segmentation fault. This was solved by:

* Fixing a corrupt gcc by installing gcc 4.7 by following <http://www.rpiblog.com/2014/07/installing-gcc-on-raspberry-pi.html>
* Fixing the python2.7 installation by issuing ‘sudo apt-get -y remove libpython2.7 python-dev python2.7-dev’ and then ‘sudo apt-get -y install libpython2.7 python-dev python2.7-dev’.

After this the OpenCV works on the Raspberry Pi!

## OpenCV attempt 3 (and final): installing the OpenCV libraries and python-opencv

The simplest way to install OpenCV is like it is described in <http://petrkout.com/electronics/low-latency-0-4-s-video-streaming-from-raspberry-pi-mjpeg-streamer-opencv/> and on <https://www.piborg.org/diddyborg/install>: sudo apt-get install libopencv-dev python-opencv python-numpy. The standard openCV libraries are installed together with python-opencv (library of Python bindings). Advantage is straightforward package installation. Disadvantage is that not the newest version will be installed.

In addition the following is needed to make it work:

* Fixing a corrupt gcc by installing gcc 4.7 by following <http://www.rpiblog.com/2014/07/installing-gcc-on-raspberry-pi.html>
* Fixing the python2.7 installation by issuing ‘sudo apt-get -y remove libpython2.7 python-dev python2.7-dev’ and then ‘sudo apt-get -y install libpython2.7 python-dev python2.7-dev’.

After this the OpenCV works on the Raspberry Pi!

## OpenCV pattern matching

To let the robot find its way Python-OpenCV is used to recognize specific objects. As input OpenCV reads the http MJPEG stream as a sequence of jpg frames. These are analyzed and the result is drawn back into the image, for example as a rectangle around the recognized object.

The sequence of processed images is converted to a mpeg-4 video using MEncoder. This video can be uploaded to Google Drive.

The following alorithms have been investigated:

* Finding patterns with cv2.matchTemplate().  
  This works by finding a match of a template image within the input image. This works quite well but is CPU intensive and only one template is matched at a time which is not very practical for robot navigation.
* Finding contours with cv2.findContours(). This can be done by first thresholding the picture on colors with cv2.inRange() and then finding the contours. This works well, but the difficulty is in finding a good color range. This way for example it is possible to detect for blue shapes but the performance is very dependent on the lighting conditions.
* Finding blobs with cv2.SimpleBlobDetector(). This works on a grayscale input image. Blob filtering can be done by color, size and shape. This way for example circles can be searched. As the blob detection algorithm itself converts the grayscale image to a binary image using different threshold levels the performance is quite independent of the lighting conditions. Up till now this is the best performing algorithm which is useful for robot navigation. Performance on a Rasberry Pi 2B is about 2 frames per second on a 320 \* 240 image.

## OpenCV motion detection

To detect motion cv2.absdiff() is used to determine the difference between two grayscale images. The resulting difference image is then converted into a black-white image using cv2.threshold(). The nonzero pixels are counted and are used to determine whether there is motion or not. To make the detection robust the motion is considered to be true motion if it occurs three images in sequence.

# Arduino Uno configuration

# Measurements

The measurements below were done using a digital Multimeter Dynatec 5010C.

The current measurements were done in the 10A current mode, and the terminals in series with the battery wires.

|  |  |  |
| --- | --- | --- |
| Voltage measurements with 6xAA batteries, batteries appr. 80% charged | | |
| Measurement | **Value** | **Remark** |
| Voltage before regulator, no current | 7.85 V | Measured at battery terminal |
| Voltage before regulator, current 320 mA | 7.55 V (7.51V) | Measured at battery terminal (measured at regulator) |
| Voltage before regulator at full current; 4 motors at max. speed | 7.15 V | Measured at regulator |
| Voltage after regulator, current 320 mA | 4.97 – 4.99 V | Measured at regulator |
| Voltage after regulator at full current; 4 motors at max. speed | 4.96 – 4.98 V | Measured at regulator |

|  |  |  |
| --- | --- | --- |
| Current measurements with 6xAA batteries, batteries appr. 80% charged | | |
| Measurement | **Value** | **Remark** |
| Raspberry Pi B+ only, no Wifi dongle, no camera | 220 mA | Raspberry Pi 2B uses about 30 mA or more extra dependent on how many cores are active. |
| Raspberry Pi only + Wifi dongle, no camera | 320 mA |  |
| Raspberry Pi only + Wifi dongle + camera | 450 - 500 mA | Current varies |
| + Arduino + motor shield | + 40 mA | Delta current measured |
| + 4 motors at speed 100 | + 400 mA | Delta current measured |
| + 4 motors at full speed 255 | + 780 mA | Delta current measured |
| + 4 motors at full speed 255 and all blocking | + 2180 mA | Delta current measured |
| + Hitec HS-422 servo at rest | +8 mA | Servo library pin attach or detach does not influence rest current , when pin is attached servo position is ‘fixed’ |
| + Hitec HS-422 servo and blocking | +500 mA |  |
| Total current with Raspberry Pi + Wifi dongle, no camera | Appr. 360 mA | Rest mode |
| Total current with Raspberry Pi + Wifi dongle + camera + 4 motors at full speed 255 | Appr. 1250 mA | Action mode |
| Total max. current with Raspberry Pi + Wifi dongle + camera + 4 motors at full speed 255 and all blocking | Appr. 2500 mA | Max. action mode |
| Light on (no other devices connected) | 380 mA | Two flashlights in series, 9 leds per flashlight |

|  |  |  |
| --- | --- | --- |
| Video delay measurements | | |
| Measurement | **Value** | **Remark** |
| No motors and using camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 10 --exposure auto --hflip yes --vflip yes | Appr. 0.1 s | Connection to laptop with local Wifi |
| 4 motors at full speed and using camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 10 --exposure auto --hflip yes --vflip yes | Appr. 0.1 s | Connection to laptop with local Wifi |
| No motors and using camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 30 --exposure auto --hflip yes --vflip yes | Up to 3 seconds. This delay is mainly in the network and not on the Raspberry; when pulling out the Wifi dongle the Raspberry the video on the Galaxy is continuing for about 2 seconds! | Connection to Galaxy S4 phone on 3G network, framerate 30 (to get more delay) |

|  |  |  |
| --- | --- | --- |
| Video bitrate measurements over Wifi | | |
| Measurement | **Value** | **Remark** |
| Camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 10 --exposure auto --hflip yes --vflip yes | 4 Mbit/s, dependant of content | 1 connection open, measured using the ‘wavemon’ tool |
| Camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 10 --exposure auto --hflip yes --vflip yes | 8 Mbit/s. dependant of content | 2 connections open, measured using the ‘wavemon’ tool |
| Camera settings: --encoding mjpeg --width 800 --height 600 --quality 10 --framerate 2 --exposure auto --hflip yes --vflip yes | 1 Mbit/s, dependant of content | 1 connection open, measured using the ‘wavemon’ tool. This setting can be used when connection is poor. |

# Operate the DFRobot using only 3G or 4G

Would it not be nice to be able to operate the DFRobot using only 3/4G? This makes it possible to operate the DFRobot in a remote area where only 3/4G is available, so virtually all over the world. No Wifi required! We will use a Samsung Galaxy S4 phone with a KPN Mobile 3/4G subscription for this experiment and a MacBook in a home network behind a router.

**The first step:**  
Without Wifi it is still possible to operate the DFRobot with an additional phone close to the DFRobot, using the phone’s Wifi tethering capabilities to connect to the DFRobot. Opening the phone’s browser the DFRobot webpage can be seen at <tethering IP address>:44444, for example 192.168.43.182:44444. This way the DFRobot can be operated from the Samsung phone provided the phone is close to the DFRobot.  
**The problem:**  
So far so good, but how can the DFRobot be accessed from any other location through the standard internet? Because the Samsung phone is connected to 3/4G through a provider like KPN Mobile it has no own IP address; when checking the IP address with <http://www.whatsmyip.org/> it will show a KPN Mobile IP address which is not directly related to the phone. This means the DFRobot cannot be accessed from the internet as such.  
It may be that when IPv6 is rolled out phones will get a unique public IP address which can be reached directly, but this can take a couple of years or more.  
**The solution:**Though the Samsung phone connected to 3/4G cannot be reached from the internet, the phone itself cán reach the internet! So the phone could first make a connection to the MacBook. Then a ‘tunnel’ is created which is used to let the MacBook access the DFrobot webserver through a standard browser. This can be accomplished by creating a so-called ‘SSH reverse tunnel’. It is called ‘reverse’ because the DFRobot phone has to initiate it but the tunnel itself is used in the other direction: from the MacBook running a browser and sending requests to the webserver running on the DFRobot. What happens is that first the Samsung phone sets up a standard SSH connection to the MacBook. Then an extra service is started to forward all traffic on port X from the MacBook back to port Y on the DFRobot phone. This is the ‘reversed’ tunnel. This technique is commonly used, for example to bypass firewalls.  
**Steps to set this up:**

* The Raspberry Pi must be able to connect to the Wifi hotspot of the Samsung phone. For this the Wifi credentials of the hotspot connection must be added to ‘/etc/wpa\_supplicant/wpa\_supplicant.conf’.
* On the router, assign a static IP address like 192.168.1.55 to the MacBook. This is needed because we need to forward the SSH port to the MacBook.
* On the router open SSH port 22 to the MacBook.
* On the Samsung phone install ‘ConnectBot’ which is an open source SSH client.
* In ConnectBot create an SSH connection to the MacBook.
* In ConnectBot add two port forwards to this connection:
  + Forward 1, type remote: from port 44444 (MacBook) to 192.168.43.182:44444 (Samsung phone)
  + Forward 2, type remote: from port 44445 (MacBook) to 192.168.43.182:44445 (Samsung Phone)

Two forwards are needed because the DFRobot webserver uses port 44444 for the html page and port 44445 for the MJPEG stream.  
In this case 192.168.43.182 is the address issued by the Samsung Wifi hotspot.

* On the Samsung phone activate Wifi hotspot. The Samsung phone will have the IP address 192.168.43.1 (hardcoded in Android).
* Now put the DFRobot and the Samsung phone in a remote area with 3/4G only.
* On the MacBook, open localhost:44444 in the browser and voilà!

**Experience:**The setup described works quite nice using a Samsung Galaxy S4 with a KPN Mobile 3/4G subscription. The DFRobot which is outside in a 3/4G area can be controlled from a MacBook using a standard browser. Controlling the DFRobot is about as fast as with a Wifi setup. The video performance might be less: the 1 second latency of the MJPEG stream became around 3 seconds in a specific situation (inside) but will obviously depend on the quality of the 3/4G connection. Otherwise the 3/4G experience is the same as the Wifi experience!

# Circuit diagram

# MacBookPro:Users:fhict:ReneB:GitHub:rbakx:DFRobot:DFRobotCircuit.png

