**MyExoMy project 2021**





Rene Bakx, May 2021

Contents

[1. Introduction 4](#_Toc74120557)

[2. Connecting to the Raspberry Pi with SSH and remote desktop (VNC). 4](#_Toc74120558)

[2.1. For the first time 4](#_Toc74120559)

[2.2. When connection is lost for whatever reason, and no monitor connected to the Raspberry Pi is available. 4](#_Toc74120560)

[3. Way of working 4](#_Toc74120561)

[3.1. For development 4](#_Toc74120562)

[3.2. For normal operation 5](#_Toc74120563)

[3.3. About Docker 5](#_Toc74120564)

[3.4. Installing ROS Melodic on Raspberry Pi OS (Debian Buster), no Ubuntu or Docker. 5](#_Toc74120565)

[4. From Exomy to MyExoMy 5](#_Toc74120566)

[4.1. Additions / changes in hardware 6](#_Toc74120567)

[4.2. Additions / changes in software 7](#_Toc74120568)

[5. Topics 8](#_Toc74120569)

[5.1. Wifi configuration in /etc/wpa\_supplicant/wpa\_supplicant.conf 8](#_Toc74120570)

[5.2. Wifi performance 8](#_Toc74120571)

[5.2.1. Video stream buffering 8](#_Toc74120572)

[5.2.2. External antenna for the Raspberry Pi 9](#_Toc74120573)

[5.2.2.1. External USB Wifi dongle AWUS036ACM 9](#_Toc74120574)

[5.2.2.2. External Antenna connected to RPi Wifi module 10](#_Toc74120575)

[5.2.3. Access points and roaming 10](#_Toc74120576)

[5.3. Accessing the ExoMy from outside the LAN 10](#_Toc74120577)

[5.4. Communication between the Docker container and the Raspberry host 10](#_Toc74120578)

[5.5. Low Power ATmega328 11](#_Toc74120579)

[5.6. Start and view RTSP stream 11](#_Toc74120580)

[5.7. Improved streaming to web page 11](#_Toc74120581)

[5.7.1. Increase resolution in launch/exomy.launch 12](#_Toc74120582)

[5.7.2. MJPEG-Streamer 12](#_Toc74120583)

[5.7.3. GStreamer 12](#_Toc74120584)

[5.7.4. Pi H264 To Browser 13](#_Toc74120585)

[5.7.5. UV4L 13](#_Toc74120586)

[5.7.5.1. MJPEG encoding 13](#_Toc74120587)

[5.7.5.2. H.264 encoding 14](#_Toc74120588)

[5.7.6. Rpi-WebRTC-Streamer 14](#_Toc74120589)

[5.7.7. EasyRTC 15](#_Toc74120590)

[5.7.8. Jitsi on Raspberry Pi 16](#_Toc74120591)

[5.7.9. Chrome WebRTC 16](#_Toc74120592)

[5.8. SSL / TLS certificates 17](#_Toc74120593)

[5.9. Power Board 17](#_Toc74120594)

[5.10. EasyEda 17](#_Toc74120595)

[5.11. EasyEda PCB settings: 17](#_Toc74120596)

[5.12. Ultimaker Cura settings 18](#_Toc74120597)

[5.13. Low Power 18](#_Toc74120598)

[5.13.1. Default EEPROM bootloader configuration settings, low power settings in red. 18](#_Toc74120599)

[6. Measurements 19](#_Toc74120600)

[6.1. Current measurements 19](#_Toc74120601)

[6.2. Video bitrate and CPU load measurements 19](#_Toc74120602)

[7. Useful links 21](#_Toc74120603)

# Introduction

All credits for this document go to Maximilian Ehrhardt and Miro Voellmy who created a 3D printable robot ExoMy, a small version of the ExoMars Europe’s Rosalind Franklin ExoMars rover which is scheduled to be launched to Mars in 2022.  
They made a very nice open source hardware and software design. This document describes how I created my own version of the ExoMy, the MyExoMy.

This document can be seen as a notebook with practical information mainly for personal use.

# Connecting to the Raspberry Pi with SSH and remote desktop (VNC).

## For the first time

This described at <https://github.com/esa-prl/ExoMy/wiki>, but not for headless mode (without a monitor). This is described at <https://desertbot.io/blog/headless-raspberry-pi-4-remote-desktop-vnc-setup>.

## When connection is lost for whatever reason, and no monitor connected to the Raspberry Pi is available.

If for whatever reason connection to the Raspberry Pi is lost, the most obvious thing to do is to attach a monitor to the Raspberry Pi and fix it. If however, no monitor is available, remove the SD memory card from the Raspberry Pi and put it in the laptop. If the SD card is not corrupt, two drives are mounted, one of them is the boot partition and can be read with Windows. Follow again the instructions on <https://desertbot.io/blog/headless-raspberry-pi-4-remote-desktop-vnc-setup>. Make sure all settings are done correctly the first time the SD card is put back in the Raspberry Pi. Note that the empty ‘ssh’ file and the file ‘wpa\_supplicant.conf’ will be removed again after the next reboot. So if things do not work the process has to be repeated.

# Way of working

## For development

* Connect to the RPi using SSH.
* Use sh ~/ExoMy\_Software/docker/run\_exomy.sh -d to start the ExoMy Docker container for development. This will map the Docker container folder /root/exomy\_ws/src/exomy which contains the ExoMy scripts to /home/pi/ExoMy\_Software. This is done by the run\_exomy.sh script by using the ‘-v’ (volume) option with the docker run command.
* On the host computer open FileZilla and connect to the RPi using port 22 for SFTP (SSH FTP). This way files can be transferred between the host computer and /home/pi/ExoMy\_Software. Because this folder is mapped to the ExoMy Docker folder /root/exomy\_ws/src/exomy any change will have immediate effect on the running container. This way the changes will be permanent and not lost when the container is stopped.
* When changing something in the GUI Web Interface, like the index.html or style.css, it can be necessary to delete the history of the browser at the host computer before the changes can be seen!
* After logging in to the RPi one can run the Docker exomy\_devel container for development with:  
  sh ~/ExoMy\_Software/docker/run\_exomy.sh -d : starts the exomy\_devel container  
  And then in sequence:  
  source /opt/ros/melodic/setup.bash : does some catkin setup   
  cd /root/exomy\_ws  
  catkin\_make : builds all ROS packages  
  http-server --ssl --cert /root/exomy\_ws/src/exomy/ssl/server.crt --key /root/exomy\_ws/src/exomy/ssl/server.key src/exomy/gui -p 55555 & : starts the web server  
  source devel/setup.bash : does some catkin setup  
  roslaunch exomy exomy.launch : starts the ROS nodes as specified in exomy.launch
* When a python script is changed it is required to quit ROS and to issue roslaunch again.
* In docker/run\_exomy.sh one can see that all Docker containers are based on the same image named ‘exomy’.

## For normal operation

* Use a single sh ~/ExoMy\_Software/docker/run\_exomy.sh -a and ExoMy will become operational, also after a reboot.
* The ExoMy can be operated through its web page at port 55555.

## About Docker

Docker is used on the ExoMy to provide a reliable and reproducable runtime environment. Docker uses a stable read-only image (ros:melodic) and additional installation commands specified in docker/Dockerfile are used to build a container where the software runs. Compare this with using an writable image and manually adding installations to it and letting it grow and grow. This is very hard to maintain or to roll back or repair if somethings goes wrong.

The ExoMy docker containers are based on Ubuntu 18.04.5 LTS (Bionic Beaver).

Advantages Docker

* Provides a reliable and reproducable runtime environment.

Disadvantages Docker

* Some high performance tools do not work inside the Ubuntu Docker environment or with less performance, like MJPG-Streamer (less performance) and UV4L (Raspberry Pi OS only).

## Installing ROS Melodic on Raspberry Pi OS (Debian Buster), no Ubuntu or Docker.

To avoid the Docker / Ubuntu disadvantages one can install ROS on the Raspberry Pi OS (Debian Buster) following <https://www.instructables.com/ROS-Melodic-on-Raspberry-Pi-4-RPLIDAR/>.

# From Exomy to MyExoMy

The standard open source hard and software design is a very good basis and suitable for all kinds of additions. Below the additions and changes are listed.

## Additions / changes in hardware

* A 20W solar panel is added to the design to charge the batteries. This way the ExoMy can be 24/7 operational without the need of a separate charging station. Not that ‘operational’ means that for most of the time the ExoMy will be sleeping and / or charging. Even in winter with dark and short days one should be able to drive a few minutes every day.  
  In summertime this is about 20 minutes every day. This could be higher but the charging current is limited to 60 mA to keep things simple and to be sure the batteries are not overcharged, although charging will stop when the batteries reach a threshold voltage.
* To accommodate the charging with the solar panel a Power Board is designed. This board provides safe charging for the batteries and power to all components like the Raspberry Pi, servo board and the lights. It can also put the ExoMy in deep sleep.
* Instead of the LiPo accupack 11.1 V 3000 mAh, 5x baby C batteries HR14 NiMH 1.2 V 5500 mAh are used. This because of the following reasons:
  + The charging of NiMH batteries with a solar panel is more straightforward and safer.
  + No voltage converters (buck converters) are needed. Although these can reach a efficiency of > 90%, in practical use efficiency can be as low as 60%. Instead the 5x 1.2V batteries can directly provide 6V for the servo’s. A low drop voltage regulator is used to provide 5.0V for the Raspberry Pi.
  + Using a branch at the fourth battery provides 4.8V. This is needed to power an ATmega328P processor which can put the ExoMy in deep sleep using as little as 100nA. This cannot be accomplished using a LiPo and an voltage converter.

There is also a disadvantage of the NiMH batteries. Although the energy content is about the same as for the LiPo battery, the weight is about 120 grams more.  
  


Figure 1: Power board, created with EasyEDA



Figure 2: Power board PCB, manufactured at JLPCB

* Two headlights are added so ExoMy can also see at night.
* Rubber 4 mm diameter O-rings are placed around the wheels. This to give ExoMy a smooth ride on hard surfaces. On soft surfaces like sand the O-rings sink into the ground and the teeth of the wheels will provide the grip.

## Additions / changes in software

* The gui/index.html and gui/style.css were adapted to have a bigger video image and to display some statuses like battery, solar panel and wifi status.
* The ExoMy webserver port is changed from 8000 to 55555 in docker/run\_exomy.sh and docker/entrypoint.sh, just to have a less obvious port.
* In docker/Dockerfile some additional installations are done:
  + SMBus needed for I2C to communicate with the ATmega328P processor. This script needed smbus to be installed.
  + OpenSSH for communication between the Docker container and the Raspberry Pi host.
* In docker/entrypoint.sh some lines are added to copy the SSH key pair to /root for authentication so SSH can be used without password by a Python script.
* On the Raspberry Pi host the public half of the SSH key pair is installed manually on the Raspberry Pi host. See <https://upcloud.com/community/tutorials/use-ssh-keys-authentication> for a description.
* In src/robot\_node.py additional code was added to publish some statuses like battery status, solar panel status and Wifi status to the ‘/battery\_status’, '/solarpanel\_status' and wifi\_status topic respectively every second.
* In src/robot\_node.py additional code is added to handle the messages published by the Web page (in index.html) on the ‘/own\_button’ topic. These messages are:
  + lights\_on. Through I2C the ATmega328P is instructed to switch on the lights.
  + lights\_off. Through I2C the ATmega328P is instructed to switch on the lights.
  + goto\_sleep. Through I2C the ATmega328P is instructed to go to sleep. This will be acknowledged by the Arduino. Only after the acknowledge the Raspberry Pi will be shut down. The Docker container uses SSH to send the shutdown command to the Raspberry Pi host. The sequence of events:
    - Sleep button is pressed on the web page.
    - In index.html the message ‘goto\_sleep” is published on topic ‘/sleep\_status’.
    - In robot\_node.py the message callback function sends through I2C the instruction to the ATmega328P to go to sleep.
    - The ATmega328P acknowledges the goto\_sleep and goes to sleep after a delay. This delay is needed to enable the Raspberry Pi to shut down in a proper way.
    - robot\_node.py checks for the acknowledge and then sends the shutdown command with SSH to the Raspberry Pi host.
* In src/motor\_node.py the watchdog\_timer for stopping the drive motors is changed from 5 s to 0.5 s. This to make sure that if the Wifi connection drops the ExoMy stops in 0.5 second. It still is possible that after the ExoMy stops it starts moving again due to delayed Wifi packet delivery. A way to deal with this is to release the joystick on the web page as soon as the video hangs. Tests show that the ExoMy then indeed will stop within 0.5 s.
* In src/motors.py the drive motors are switched off when the speed is very low. This to eleminate the need to calibrate the drive motors to zero speed using the potentiometer every time.

# Topics

Below some specific topics are described, just to document things which took some effort to find out.

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

On the Raspberry Pi host use sudo nano /etc/wpa\_supplicant/wpa\_supplicant.conf to add the wifi SSID and password of the network to which ExoMy can connect.

## Wifi performance

### Video stream buffering

When driving around connection hiccups can occur. Some observations:

* Hiccups occur also when the received signal strength is still good (e.g. -60 dBm).
* Hiccups occur mostly when moving, when standing still the signal mostly is ok.
* The video bitrate does not seem to have a big influence on the hiccups. Even with low video bitrates (e.g. 1 Mbit/s) the hiccups occur.
* Hiccups can be seen in Wireshark as TCP retransmissions. Note that with WebRTC UDP is used which does not care about retransmissions which gives a better video streaming experience.
* It seems that the problem is Adjacent Channel Interference. This happens mainly when the ExoMy is moving in a location where there are many other Wifi channels and the own Wifi channel does not really standout as it would inside close to its own access point.
* The connection bitrate measured with Windows Task Manager -> Performance -> Wi-Fi shows drops toward 0 Mbit/s during a hiccup.

Measurements using Windows Task Manager -> Performance -> Wi-Fi show that during a connection loss video is buffered at the streaming server side. As soon as the connection is restored the video buffer is sent resulting in a peak in the bitrate as can be seen in the figure below.



Figure 3 Video buffering at streaming server side

When the HTML video element is used, the video received is also buffered and thus causing the latency to build up after each connection hiccup.  
When the HTML video element is used, this can be solved at the receiver side in Javascript by setting the video attribute currentTime to its maximum value when the buffer built up gets too large. This will effectively jump the video forward to the current time.  
vid.currentTime = Number.MAX\_VALUE;  
With an MJPEG stream the HTML img element is used and then this is no issue. Also when WebRTC is used this is no issue.

To monitor the network bandwidth from the Raspberry Pi side from the terminal, install vnstat and use vnstat -i wlan0 --live.

### External antenna for the Raspberry Pi

Because the Raspberry Pi onboard Wifi antenna is low to the ground and covered by the servo board, an external Wifi antenna or USB Wifi dongle can be used.

#### External USB Wifi dongle AWUS036ACM

To disable onboard Wifi and also Bluetooth (to avoid interference) the following lines are added to /boot/config.txt:  
#dtoverlay=disable-wifi  
#dtoverlay=disable-bt  
The dongle will become wlan0 then.

In case a dongle is used do not forget to change the MAC address in the router for the static DHCP address setting for the ExoMy. This should be the MAC address of wlan0.

However the results of the AWUS036ACM were disappointing. The received signal strength in dBm was ok but there were many hiccups for no apparent reason.  
Am disadvantage of an external Wifi dongle is the additional power it uses (around 300 mA for the AWUS036ACM).

#### External Antenna connected to RPi Wifi module

Another possibility is to connect an external antenna to the Raspberry Pi following  
<https://www.youtube.com/watch?v=MTwWnZG8wUY&ab_channel=CalebBeglyFun%26Tech>.  
The results are quite good, the received signal strength improves about 5 dBm.

Figure 4 External antenna with an U.FL connector on the Raspberry Pi board

### Access points and roaming

When driving around outside there can be quite some hiccups in the connection / video, sometimes up to 5 seconds! Some observations:

* Serious hiccups occur with an Access Point handover. This also is the case with a Wifi Mesh network.
* The best solution found so far is to have one access point at the center of ExoMy’s playground. This prevents handovers.
* 5 GHz seems to have better performance than 2.4 GHz probably due to the fact that the 5 GHz band is less crowdy.

## Accessing the ExoMy from outside the LAN

For accessing the ExoMy from outside the LAN it is convenient to give the Raspberry Pi a fixed IP address so port forwarding can be used. At the same time we want DHCP to be used on the ExoMy so it can also be used in other networks like in a phone’s tethering network. This can be done by letting the router assign a static DHCP IP address (192.168.1.42) to the ExoMy, using its MAC address.  
For the ExoMy three ports must be forwarded:

* ExoMy webserver port: 55555 (TCP)
* ExoMy websocket port: 9090 (TCP)
* ExoMy WebRTC port: 8080 (TCP + UDP)

## Communication between the Docker container and the Raspberry host

For some purposes communication between the Docker container and the Raspberry Pi host is needed. For example to shut down the Raspberry Pi host or to get the Wifi status. This is accomplished by using SSH. To enable Python to use SSH, SSH keys must be (one time) manually installed in the Docker container (private and public) and on the Raspberry Pi host (public) with:

mkdir -p ~/.ssh  
chmod 700 ~/.ssh  
ssh-keygen -t rsa

And finally copy the public key to the Raspberry Pi host. On Docker for Linux, the IP address of the gateway between the Docker host and the bridge network is 172.17.0.1 if you are using default networking. So be sure to use that address here.

ssh-copy-id -i ~/.ssh/id\_rsa.pub pi@172.17.0.1

When the container is restarted, the SSH keys in the container will be lost, zo the are copied back by some added lines the docker/entrypoint.sh script. The public SSH key on the Raspberry Pi host will remain.  
The .ssh folder containing the public and private key is added to the .gitignore file.  
See also <https://upcloud.com/community/tutorials/use-ssh-keys-authentication> for a description.

## Low Power ATmega328

For the MyExoMy project the ATmega328P-PU is used with an alternative bootloader which sets the clock to the 8 MHz internal clock. The new board definition (named 'ATmega328 on a breadboard (8 MHz internal clock)') and bootloader is available in the breadboard-1-6-x folder.

See <https://www.arduino.cc/en/Tutorial/BuiltInExamples/ArduinoISP> for how to burn a bootloader on one Arduino board using another Arduino board as ISP.

The 'ATmega328 on a breadboard (8 MHz internal clock)' board is made available to the Arduino IDE by copying the breadboard folder to C:\Program Files (x86)\Arduino\hardware.

See <https://www.arduino.cc/en/Tutorial/BuiltInExamples/ArduinoToBreadboard> for a description of how to connect an ATmega328P on a breadboard and use it.

To make the new board available in Visual Studio Code an Atmega328\_on\_breadboard\_8MHz.json file is added to C:\Users\reneb\.platformio\platforms\atmelavr\boards. This json file is created by copying the uno.json file (which is for the Arduino Uno) and adapted the content with info from boards.txt in breadboard-1-6-x.

## Start and view RTSP stream

* Start RTSP server on Raspberry Pi:  
  raspivid -o - -t 0 -hf -w 1920 -h 1080 -fps 30 | cvlc -vvv stream:///dev/stdin --sout '#rtp{sdp=rtsp://:8554/x}' :demux=h264
* View in VLC:  
  GUI -> Media -> Open Network Stream -> rtsp://192.168.1.42:8554/x

Note: it is needed to give the stream a name, here ‘x’.

## Improved streaming to web page

To improve the video streaming several streaming options have been investigated. At the moment Rpi-WebRTC-Streamer is used. This because the combination of H.264 encoding and WebRTC (UDP) gives the best performance.  
In all cases, when using the Raspberri Pi camera, choose a resolution such that is used so the FoV (Field of View) of the Raspberry Pi camera is full, see <https://picamera.readthedocs.io/en/release-1.12/fov.html>.

When the streaming is done from the Raspberry Pi host, the web\_video\_server and usb\_cam ROS nodes must be removed from the launch/exomy.launch file. Also the port forwarding of port 8080 must be removed from the docker/run\_exomy.sh file.

### Increase resolution in launch/exomy.launch

The most straigthforward way is to increase the resolution of the usb\_cam ROS node as specified in launch/exomy.launch. This can be set to 800x600 and maintaining an acceptable latency (< 0.5 s).

### MJPEG-Streamer

The MJPG-Streamer can be installed in the container following <https://www.sigmdel.ca/michel/ha/rpi/streaming_en.html>. The installation is copied to /root/exomy\_ws/src/exomy which is shared with the host This way the MJPG-Streamer will be available in a new container.  
The following command can be used to stream the video stream to a web page:

/root/exomy\_ws/src/exomy/mjpg-streamer/mjpg\_streamer -i "/root/exomy\_ws/src/exomy/mjpg-streamer/input\_uvc.so -n -f 20 -q 85 -r 1200x900" -o "/root/exomy\_ws/src/exomy/mjpg-streamer/output\_http.so -p 8080 -w /root/exomy\_ws/src/exomy/mjpg-streamer/www"

This command will use /dev/video0 as the video device.

The stream can be viewed via the MJPG-Streamer web page at 192.168.1.42:8080 or directly at 192.168.1.42:8080/stream\_simple.html.

To stop the stream use:

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

On the web page the stream can be picked up using  
stream\_url = '//' + host\_url + ':8080?action=stream'

**Advantages of MJPG-Streamer**

* Reasonable image quality and low latency (< 0.5 s @ 1920x1080).

**Disadvantages of MJPG-Streamer**

* Not maintained anymore.
* When used with input\_uvc.so i.s.o. input\_raspicam.so the image quality and latency are worse.

### GStreamer

Install GStreamer on the Raspberry Pi host following <https://raspberry-projects.com/pi/pi-hardware/raspberry-pi-camera/streaming-video-using-gstreamer>

On Linux it can be installed following <https://gstreamer.freedesktop.org/documentation/installing/on-linux.html>

Basic test when on the desktop (directly or with VNC Viewer):  
gst-launch-1.0 v4l2src device="/dev/video0" ! videoconvert ! autovideosink  
When run on the desktop (not via SSH) one should see a video window.

Start streaming with:  
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-raw,width=640,height=480,framerate=30/1 ! videoconvert ! jpegenc ! rtpjpegpay ! tcpserversink host=192.168.1.42 port=8080

**Advantaged GStreamer**

* GStreamer is open source and multi-platform.

**Disadvantages GStreamer**

* Most examples show a GStreamer server streaming to a GStreamer client. A working example of streaming to a web page has not been found yet.

### Pi H264 To Browser

A custom H264 streamer making use of Tornado and jMuxer.  
**Advantages Pi H264 To Browser**

* The video is very customizable, the quality setting is useful.

**Disadvantages Pi H264 To Browser**

* The package makes use of python3-picamera, which cannot be installed in the Ubuntu Docker container.
* When running, the video starts at low latency (< 0.5 s), but the latency increases up to a few seconds. This still has to be solved.

See <https://github.com/dans98/pi-h264-to-browser>.

### UV4L

UV4L can be installed on the Raspberry Pi host following  
<https://www.linux-projects.org/uv4l/installation/>

#### MJPEG encoding

A uv4l\_raspicam streaming service can be started with:  
sudo systemctl start uv4l\_raspicam  
It will use then the settings as specified in /etc/uv4l/uv4l-raspicam.conf.

To make it start after reboot:  
sudo systemctl enable uv4l\_raspicam

The service will use the /dev/video0 device.  
The service will be restarted automatically after a reboot.

To stop the service  
sudo systemctl stop uv4l\_raspicam  
To disable the service (so it will not start after the next reboot):  
sudo systemctl disable uv4l\_raspicam  
to check the status:  
sudo systemctl status uv4l\_raspicam

The uv4l streaming can be also started manually with:  
uv4l --auto-video\_nr --driver raspicam --encoding mjpeg --width 1200 --height 900 --framerate 20 –quality 85 --server-option '--port=8080'  
The uv4l process is a process without tty (daemon) and can be listed with  
ps ax  
The stream can be stopped by killing the process.

On the web page the stream can be picked up using  
stream\_url = '//' + host\_url + ':8080/stream/video.mjpeg'

#### H.264 encoding

It is also possible to use H.264 encodeing from the Rasberry Pi by issuing  
uv4l --auto-video\_nr --driver raspicam --encoding h264 --width 1200 --height 900 --framerate 20 –quality 85 --server-option '--port=5000'

The stream can be viewed under Windows with VLC installed by issuing from the command prompt:  
"c:\Program Files (x86)\VideoLAN\VLC\vlc.exe" [http://192.168.1.42:5000/stream/video.h264 --demux h264](http://192.168.1.42:5000/stream/video.h264%20--demux%20h264)

The video is of high quality, the framerate and the bit rate of about 10 Mbit/s instead of 100 Mbit/s when mjpeg is used (all other conditions are the same).  
However, when watching with VLC the latency is about 3 seconds and the processing (probably on the Raspberry Pi) seems to be a problem as sometimes the video freezes.  
To display the stream on a web page it should be packed into mp4 first. This has not been investigated yet.

**Advantages of UV4L**

* UV4L uses WebRTC (Web Real-Time Communications). WebRTC started in 2011, but in 2021 it became an official Web Standard and a World Wide Web Consortium (W3C) recommendation.
* Using WebRTC with H.264 video encoding the video quality is high (when hardware encoding and decoding is enabled) and latency is consistently low (< 0.5 s @ 1920x1080) which gives a great driving experience.
* UV4L uses WebRTC which is designed for low latency connections. WebRTC uses UDP. For comparison, websockets use TCP. While driving UTP gives a better experience; after a connection hiccup the video takes up again fast.

**Disadvantages of UV4L**

* UV4L is closed source. Therefore it is difficult to customize / integrate with own web page.
* Does not seem to work on Android phone or tablet when hardware encoding is used (H.264). When no hardware deconding is selected (VP8) it works but then the resolution is low and cannot be changed.
* The connection for WebRTC must be SSL, meaning on the client device the CA certificate must be added as a trusted root authority.
* UV4L does not install on the ExoMy container Ubuntu version. Therefore it must run on the Raspberry Pi host which is not ideal.
* UV4L gives more CPU load (60%) than Rpi-WebRTC-Streamer (30%) when streaming according to htop. Note that the maximum CPU load for the Raspberry Pi 4 model B is 400% (100% for each core).
* When the streaming service runs, FileZilla is very unresponsive.

See also <https://www.youtube.com/watch?v=5QAHlZoPlgI&ab_channel=ReefNerd>.

### Rpi-WebRTC-Streamer

To start the stream: sudo systemctl start rws  
To stop the stream: sudo systemctl stop rws

**Advantages of Rpi-WebRTC-Streamer**

* WebRTC combination of H.264 encoding and UDP gives low latency and low bandwidth.
* Source code available.
* Works on desktop browser and Android browser.
* Rpi-WebRTC-Streamer gives less CPU load (30%) than Uv4L (60%) when streaming according to htop.

**Disadvantages of Rpi-WebRTC-Streamer**

* Runs only on Raspberry Pi host. Installing in a Ubuntu 18.04.5 LTS (Bionic Beaver) container gives dependency problems with libasound2. Building from source in a container might resolve this.
* There is a bug causing the webrtc-streamer to crash and stopping the stream. The error message when running sudo /opt/rws/webrtc-streamer –verbose:  
  [009:295] [22219] (mmal\_wrapper.cc:150): frame\_buf\_pos : 196657, buffer length: 11943  
  # Fatal error in: mmal\_wrapper.cc, line 153  
  # last system error: 0  
  # Check failed: (int)(frame\_buf\_pos\_ + buffer->length) < size\_  
  # Aborted  
  This seems to be related to the MMAL buffer size according to a posted debug version (<https://github.com/kclyu/rpi-webrtc-streamer/issues/37>) with increased frame buffer size (from 65536\*2 to 65536\*3):  
  (mmal\_wrapper.cc:152): frame\_buf\_pos : 131119, buffer length: 25370  
  One can see that the offset 131119 falls outside the original MMAL frame buffer size.  
  This repaired version of the webrtc\_streamer executable is installed in /opt/rws which fixes the issue.  
  It is not clear why this bug fix is not in the latest release.  
  Fixing and rebuilding from source would be an option, but the WebRTC native-code library has to be cross compiled from Ubuntu. Unfortunately the build recipe is very complicated and lots of errors occur. From the GitHub comments in “Issues” it seems that building does currently not work.

See <http://www.softwaresamurai.org/2017/10/14/uv4l-webrtc-vs-rpi-webrtc-streamer/>  
and  
<https://github.com/kclyu/rpi-webrtc-streamer/>.

### EasyRTC

**Advantages of EasyRTC**

* Runs on any platform that can run node.js.
* Good documentation on how to build your own WebRTC application!

**Disadvantages of EasyRTC**

* Current state / support is not clear as some links at <https://easyrtc.com/> are dead.  
  It seems that the project is continued at <https://github.com/open-easyrtc/open-easyrtc>.
* The demo application executed on the Raspberry Pi works but is not yet convincing; the video quality is not high and the latency is near 0.5 sec, while the bitrate is around 10 Mbit/s @800x600, 20fps. Higher resolution give a distorted image.

See <https://easyrtc.com/> and <https://github.com/open-easyrtc/open-easyrtc>.

### Jitsi on Raspberry Pi

For telepresence two way communication Jitsi can be installed on Raspberry Pi.  
prerequisite is an 64 bit OS is installed fist, like Ubuntu 64 bit and at least 4 GB of RAM is available.

See <https://peppe8o.com/self-host-your-web-meetings-with-jitsi-and-raspberry-pi/>.

### Chrome WebRTC

Chrome implements a well documented WebRTC API. An example WebRTC connection is available at <https://codelabs.developers.google.com/codelabs/webrtc-web>.

**Advantages of Chrome WebRTC**

* Uses only WebRTC functionality in the browser and a standard node.js http server as signaling server.
* Because only browser functionality is used on the Raspberry Pi, no separate programs need to be installed / maintained, besides node.js and JavaScript files. This ensures maximum compatibility with remote browsers.
* Two way video and audio supported out of the box!

**Disadvantages of Chrome WebRTC**

* At both sides a browser instance is needed, so also on the Raspberry Pi.
* On the Raspberry Pi starting Chromium from a Docker container through ssh results in a crash, unless sudo in combination with --no-sandbox is used. Because sudo is used, an additional  
  xhost si:localuser:root;  
  is needed after   
  export DISPLAY=:0 nohup;  
  in the ssh command otherwise Chromium cannot attach to a display.  
  Note that it is not needed that an actual display is attached.
* At the moment a maximum resolution of 1920x1028 (16:9) and 1440x1080 (4:3) seems to be supported in Chromium for the Raspberry Pi camera V2. This can be tested with <https://webrtchacks.github.io/WebRTC-Camera-Resolution/> or <https://test.webrtc.org/>.  
  When using this camera with Chromium and ExoMy the reported local resolution is indeed the configured one. However at higher resolutions (> 800x600) the reported resolution on the remote side varies and the video is blotchy and choppy with a high latency (> 1 s) and also the colors are distorted.  
  When testing the Raspberry Pi camera V2 on <https://webrtc.github.io/samples/> the same blotchiness and color distortion can be seen. The blotchiness and color distortion is not because of the higher CPU load. When ending the call the CPU load is low but the same artifacts can be seen in the local video stream. Also when connecting an USB camera Logitech C930e (Raspberry Pi camera V2 disabled) the video quality is good.  
  Setting the resolution to 640x480 gives reasonable results.  
  **Important note:** When changing the resolution as specified in webrtc-web/js/main.js be sure to also clear the browsing data otherwise the previous settings will remain active.
* Chrome Web RTC streaming in Chromium gives a high CPU load, see the measurements in this document. This is due to the streaming itself, not to Chromium. With the higher CPU load one has to realize that it involves two way video streaming.

See also  
<https://www.html5rocks.com/en/tutorials/webrtc/basics/>.

## SSL / TLS certificates

See <https://www.youtube.com/watch?v=e8vMTlobW3c&ab_channel=TechForum>.

The generated server.crt and server.key file can be added at the server side on the raspberry pi:

* In docker/entrypoint.sh for the exomy http-server
* In lauch/rosbridge\_websocket.launch for the ROS websocket.
* In webrtc-web/index.js for the node.js WebRTC signaling server

The generated rootCA.pem can be added in Windows Chrome to the Trusted Root Certification Authorities via Settings -> Privacy and security -> Security -> Manage certificates -> Trusted Root Certification Authorities -> Import.  
In Raspberry Pi Chromium it is something similar.

## Power Board

The power board has the following features:

* Provides 5V for the RPi and 6V for the servos and the headlights.
* Can turn the MyExoMy into deep sleep mode, consuming only 100 nA.
* Can switch on the MyExoMy using an external trigger.
* Has a light sensor to turn on the lights when it is dark.
* Has a charge connection for the batteries.
* Has a trickle charge circuit with voltage protection for the solar panel.
* Measures the battery voltage and solar panel voltage for reporting on the web page.
* Has a trigger input to wake up from deep sleep mode.

## EasyEda

* To work with EasyEda for the MyExoMy project first change the Data Directory to the corresponding EasyEda folder with EasyEda -> Setting -> Desktop Edition Setting -> Data Directory. After that with EasyEda -> File -> Open Project the MyExoMy EasyEda project can be opened.
* Switching to another project can be done by setting the Data Directory to a different folder.
* If desired the other projects can be removed from the ‘Opened Projects’ list after EasyEda -> Login and then using the right mouse button -> Refresh List.

## EasyEda PCB settings:

* The track width is chosen to be 0.8 mm with a clearance of 0.4 mm.
* The copper thickness is chosen to be 2 oz = 2x 1.4 mil = 2x 35 μm = 70 μm. Normally it is 1 oz. 2 oz is chosen to allow more current and for robustness. It cannot be set in EasyEda but it can be selected when ordering at JLCPCB.
* According to <https://www.7pcb.com/trace-width-calculator.php> with a track width of 0.8 mm and a thickness of 2 oz the current can be appr. 3A.
* Only the battery tracks have a width of 1.27 mm and with 2 oz thickness can carry appr. 5A.

## Ultimaker Cura settings

The following printer settings were changed from the default for printing the ExoMy robot with the Creality 3D CR-20 Pro 3D printer.

* Printing Temperature set to 210°.
* Build Plate Temperature set to 60°.
* Layer Height set to 0.15 mm.
* Print speed set to 50 mm/s.
* Initial Layer Speed set to 10 mm/s.
* If support is needed, Support Structure can be set to Tree for easier removal.
* If the first layer is problematic, set Build Plate Adhesion to Raft.
* For the larger parts the Build Plate Adhesion is set to None (i.s.o Skirt).

## Low Power

The Raspberry Pi can be put in low power by issuing a ‘sudo halt’. The power of the Raspberrt Pi 4 model B board will go from appr. 700 mA to appr. 16 mA in low power. For this to work the EEPROM bootloader configuration has to be adapted. This will not be used. Instead, low power will be reached using the ATmega328P processor which will completely switch off the power supply to the Raspberry Pi and the servo board.

### Default EEPROM bootloader configuration settings, low power settings in red.

See also <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711_bootloader_config.md>

BOOT\_UART=0

WAKE\_ON\_GPIO=1 -> 0

POWER\_OFF\_ON\_HALT=0 -> 1

DHCP\_TIMEOUT=45000

DHCP\_REQ\_TIMEOUT=4000

TFTP\_FILE\_TIMEOUT=30000

ENABLE\_SELF\_UPDATE=1

DISABLE\_HDMI=0 (can be set to 1 if HDMI is not used, makes no difference for low power though)

BOOT\_ORDER=0xf41

To view: rpi-eeprom-config  
To edit: sudo -E rpi-eeprom-config –edit

After editing, whether you changed anything or not, always issue a sudo reboot, otherwise the setting does not seem to have any effect.

# Measurements

## Current measurements

|  |  |  |
| --- | --- | --- |
| Current measurements, measured with battery voltage = 6.0 V. | | |
| Measurement | **Value** | **Remark** |
| Fully operational, standing still | 0.90 A |  |
| Fully operational, standing still + lights on | 1.35 A |  |
| Fully operational, driving and steering | 1.5 .. 2.0 A |  |
| Fully operational, driving and steering, lights on | 2.0 .. 2.5 A |  |

## Video bitrate and CPU load measurements

Below the measured video bitrates are shown. Some additional notes:

* Watching an average YouTube video takes a lot less. This is because the encoding is much more efficient as the latency is not an issue. It can also be seen that the bitrate comes in bursts.
* An average Teams or Zoom session takes around 3 Mbit/s.
* The ROS usb\_cam node seems to have a bitrate limit around 40 Mbit/s. The actual Wifi limit is at least 100 Mbit/s as can be seen in the table below.
* **Important note:** When measuring the video bitrates, take care not to have the VNC Viewer open, as this will add an additional bitrate!

|  |  |  |
| --- | --- | --- |
| Video bitrate over Wifi and CPU load measurements.  Notes:   * Measurements are done with the onboard with Raspberry Pi camera V2 unless noted otherwise. * The bitrates are measured with Windows Task Manager -> Performance -> Wi-Fi. * The CPU load is measured with htop. * The bitrates are measured with ExoMy outside on a cloudy day. The bitrate heavily depends on the video content and codec. Inside the scene is calmer and bitrate can be half or less if it gets dark. * The measurements done with Chrome WebRTC involve two way video streaming. * The maximum CPU load for the Raspberry Pi 4 model B is 400% (100% for each core). | | |
| Measurement | **Value** | **Remark** |
| ROS usb\_cam node  640x480 MJPEG, 10 fps, quality 50 | 15 Mbit/s | Framerate is met. |
| ROS usb\_cam node  800x600 MJPEG, 10 fps, quality 50 | 20 Mbit/s | Framerate is met. |
| ROS usb\_cam node  1920x1080 MJPEG, 20 fps, quality 85 | 40 Mbit/s | Framerate is not met, but around 4 fps. |
| Rpi-WebRTC-Streamer 1024x768 H.264, 20 fps, quality ? | 3 Mbit/s | Rpi-WebRTC-Streamer runs on Raspberry Pi host. Framerate is met. |
| Chrome WebRTC 640x480, 20fps | 1..2 Mbit/s CPU load 60..100% | Video quality ok. Remote video resolution also 640x480. Framerate is met. Latency < 0.5s. |
| Chrome WebRTC 800x600, 20fps | Appr. 2 Mbit/s CPU load 100..150% | Video a bit blotchy. Remote video resolution also 640x480. Framerate is met. Latency < 0.5 s. |
| Chrome WebRTC 1440x1080, 20fps | 2..3 Mbit/s CPU load 200..250% | Video very blotchy and choppy. Remote video resolution often switches back. Framerate is not met. Latency appr. 1 s. |
| Chrome WebRTC 1920x1080, 20fps with USB camera Logitech C930e | 2..3 Mbit/s CPU load appr. 200% | Video quality ok but choppy. Remote video resolution often switches back. Framerate is not met. Latency > 1 s. |

# Useful links

<https://www.esa.int/Enabling_Support/Space_Engineering_Technology/3D_print_your_own_Mars_rover_with_ExoMy>

<https://github.com/esa-prl/ExoMy/wiki>

<https://github.com/esa-prl/ExoMy_Software>

<https://desertbot.io/blog/headless-raspberry-pi-4-remote-desktop-vnc-setup>

<https://msadowski.github.io/ros-web-tutorial-pt1/>

<https://www.clearpathrobotics.com/assets/guides/kinetic/ros/Practical%20Example.html>

<https://learn.adafruit.com/16-channel-pwm-servo-driver>

<https://upcloud.com/community/tutorials/use-ssh-keys-authentication>

<https://dev.to/natterstefan/docker-tip-how-to-get-host-s-ip-address-inside-a-docker-container-5anh>

<https://www.sigmdel.ca/michel/ha/rpi/streaming_en.html>

<https://picamera.readthedocs.io/en/release-1.12/fov.html>