Z-WAVE.ME
BUILDS THE SMART HOME

Z-Way

User and Programmer Manual

The Z-Wave.Me Team
May 27, 2021

Contents

1 Introduction

1.1 Structure of the book

This book describes all aspects of the Z-Way controller software solution. This include both the Z-Way software solution and the hardware Z-Way runs on. The book is structured as follows:

Start » Use » Extend » Manage » Customize » Contribute

The start section provides the necessary information to fire up a Z-Way-based controller. This is followed by the explanation of the daily user interface — called Z-Way Smart Home Interface — both for standard web browser and as native app for mobile devices. The next section cover the options to extend the system by supporting more radio technologies, third-party solutions, and other applications.
The next chapter explains tools and processes to manage and troubleshoot Z-Wave networks, followed by explanations of how to customize the user interface to your specific needs.
The final section of the book is dedicated to developers and programmers who can, and are willing to, contribute to the project and/or design their own solutions based on Z-Way.
The need for technical understanding and knowledge increases from chapter to chapter.
Please note that this book will not provide any basic knowledge about the Z-Wave technology as such. Please refer to the book 'Z-Wave Essentials' as shown in figure 1.1 for an comprehensive explanation of the Z-Wave technology. The book is available at amazon.com and many other book sellers. The ISBN number is 978-1545394640.

1.2 History of Z-Way

The history of Z-Way dates back into the year of 2008. Two developers had done their own private Z-Wave controller written in Python. When they got engaged they realized that they should combine their solutions and create a second generation Z-Wave controller. The work on this merger started in May 2009 and the result - a complete Z-Wave controller written in python was certified by the Z-Wave Alliance in March 2011.

`http://products.z-wavealliance.org/products/85`

To allow porting of this code to small memory platforms the whole software was rewritten in C and Javascript was used as scripting engine. The same time the code was updated according the new Z-Wave Plus certification process and finally certified as first Z-Wave Plus compatible controller in Fall of 2014.

`http://products.z-wavealliance.org/products/1150`

After many improvements the year of 2017 brought the next major change. As first software again Z-Way in Version 3.0 supports the new innovative security architecture of Z-Wave called S2.

1.3 Status of the document

The manual is based on Z-Way software release >= 2.3.6. Some functions marked in blue text text require Z-Way v3.0.0 and up.

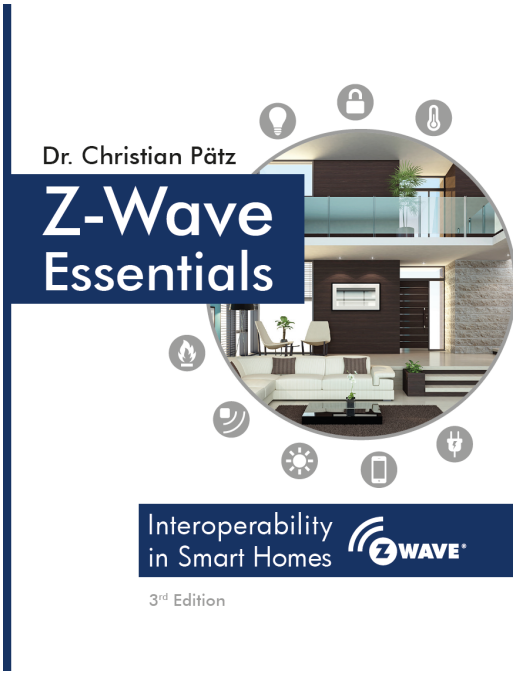


Figure 1.1: Z-Wave Essentials

2 Z-Way enabled Hardware

Z-Way is a complete software solution that is ported on various hardware. In order to run it on a certain hardware platform, the following requirements have to be met:

1. There must be a binary Z-Way distribution available for this platform. At

   `http://razberry.wave.me/z-way-server`

   you will find the most recent releases of Z-Way binary distributions for the various platforms supported:
   a) Dune HD: ARM Linux
   b) Alix-x86: Intel CPU 32 Bit Linux
   c) Contactless
   d) Debian: Intel CPU 64 Bit Linux Debian distribution
   e) Popp: For Popp Hub, Mediatek CPU, OpenWRT
   f) Raspberry Pi: for the famous Raspberry Pi, ARM based
   g) Ubuntu: Intel CPU 64 Bit Linux Ubuntu distribution
   h) Windows: For Windows Operating System

   Other platforms may be supported as well by one of these binary distributions.
2. Z-Wave transceiver connected to the platform containing a Z-Way license. Currently, the 'RaZberry Shield' always comes with an internal license enabled, while the USB Stick called UZB needs an additional license applied. Z-Way may also run on platforms with embedded Z-Wave transceivers (such as Popp HUB, Dune HD set-top box), but this requires special arrangement with the manufacturer. Please refer to the section 2.3 for more information.

Please note that Z-Way will start on a certain platform without having a Z-Wave transceiver or a licensing key. However, in this case is no support for Z-Wave. Still, this may be a good starting point to test the software free of charge. Please refer to the section 6 for possible applications usable without having Z-Wave enabled.

Z-Wave.Me currents support two basic hardware platforms with Z-Way licensing:

1. The RaZberry shield board for Raspberry Pi and compatible platforms
2. The USB Stick 'UZB' for PCs, set-top boxes, NAS, etc.

2.1 RaZberry shield board for Raspberry Pi

2.1.1 Compatibility

The RaZberry shield consists of a single PCBA with a connector to the standard GPIO pin header connector of the Raspberry Pi minicomputer. This 25-pin header connector is available on all contemporary Raspberry Pi versions, such as:

- Version A
- Version B
- Version B+
- Version 2B
- Zero
- Version 3

Even if your Raspberry Pi version is not on the list above, there is a very high chance that the shield board will work as long as your Pi has the 25-pin GPIO connector. You will find more information about the pinout of this 25-pin connector on various websites [1].

You can use other pins of the connector for other purposes as long as they do not physically conflict with the board.

2.1.2 Pinout and options on board

The RaZberry board use only four pins of this header connector:

- Gnd
- VCC (3.3V)
- Serial TX

[1] e.g. http://www.raspberry-pi-geek.de/Magazin/2015/05/Raspberry-Pi-und-Arduino-via-UART-koppeln

Figure 2.1: RazBerry on top of a Raspberry Pi

Figure 2.2: Components on RaZberry Hardware

- Serial RX

Figure 2.1 shows how the board connects to the 25-pin header on a Raspberry Pi 2
The board itself offers a few connection options as shown in Figure 2.2:

1. Raspberry Pi Connector, used GPIO pins 1-10
2. Second open connector, identical to (1)
3. Reset button
4. Open hole for a PigTail antenna. You need to break off the PCBA antenna or unsolder resistor S1 to make this work.
5. Pads to solder a uFL connector for external antenna. See

   `https://www.adafruit.com/products/1661`

   for component details. You need to break off the PCBA antenna to make this work.
6. Two LEDs for status information

The two LEDs are used to indicate the success of the boot-up self-testing and as status indicators during normal operation.

2.1.3 Boot-Up Self-Test

When powered up, the two LEDs light up, indicating that the self-testing has started. After about two seconds, they are supposed to go off indicating that that self-testing has been passed successfully. If they remain lit, this is a clear indication that the self-test failed or the device is not booting up. You will need to replace the hardware in such a case.

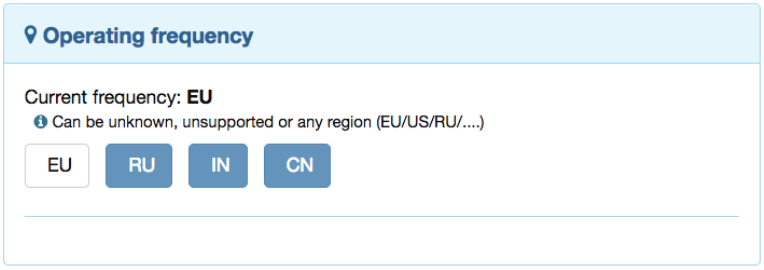


Figure 2.3: Frequency Change Option in **Z-Wave Expert User Interface**

2.1.4 LEDs during Operation

During normal operation, the two LEDs remain turned off except:

- Green LED will light up when data is transmitted.
- Red LED will light up when the Z-Wave transceiver is either in Inclusion or in Exclusion mode. Please note that these are special modes of the transceiver that block normal data communication with other nodes in the network.

2.1.5 Frequencies

The RaZberry shield itself can be tuned into every frequency used by Z-Wave. However, to protect the transceiver and Z-Wave from high energy emissions on nearby frequencies (primarily 4G/LTE cellular radios using the 852 MHz frequency band), an external antenna filter is used. This limits the frequency changes to countries that share the same antenna filter. Currently, there are three antenna filter versions identified by their SKU codes.

- SKU: ZMEEUZB2 (865…869 MHz):
  - Europe (EU)[default]
  - India (IN)
  - Russia (RU)
  - PR China (CN)
  - RSA (EU)
  - Middle East(EU)
- SKU: ZMEUUZB2 (908 ... 917 MHz):
  - All the Americas except Brazil and Peru (US) [default]
  - Israel (ISL)
- SKU: ZMEAUZB2 (919 ... 921 MHz):
  - Australia/New Zealand/Brazil/Peru/Malaysia (ANZ) [default]
  - Hongkong (HK)
  - Japan/Taiwan (JP)
  - Korea (KR)

There are two options to change the RaZberry operating frequency:

1. If you use **Z-Wave Expert User Interface** , just choose the frontend on Network » Management as shown in the figure 2.3. For more information about this **Z-Wave Expert User Interface** , please refer to chapter 7.
2. There is a shell script available at

   `http://www.z-wave.me/fileadmin/download/changezwf.sh`

   Just execute the script with

   ```
   changezwf.sh [COM Port] [US|EU|ANZ|...]
   ```

2.1.6 Certifications

RaZberry is certified for use in different countries.

CE / Europe

RaZberry complies with the new Radio Equipment Directive of the European Union in general and the EN 300 220 version 3.1.1 in particular. The device also complies with the European ROHs and REACH regulations.

Figure 2.4: USB Stick UZB

FCC / North America

The RaZberry shield was successfully tested for FCC. The FCC identifier is

2AAYUZMEURAZ.

Z-Wave Plus

The RaZberry shield is a certified hardware platform and a complete solution according to Z-Wave Plus. Please refer to the certification database

`http://products.z-wavealliance.org`

for more details.

2.2 The USB Stick UZB

The USB Stick 'UZB' allows enabling Z-Way on various platforms. Figure 2.4 shows the device.
It is plugged into a free standard USB port. UNIX-based operating systems will recognize the stick and generate a virtual serial device named **/dev/ttyACM0**
or **/dev/cu.usbmodem**
or similar. Windows will generate one virtual serial port **COM XX**
Once Z-Way is started, it will connect to the Z-Wave hardware using this virtual serial device.
The USB stick is very small (presently the smallest Z-Wave device in the world) and will stick quite close to the enclosure of the PC or NAS. This may interfere with the wireless range. If you experience problems with the wireless range, please use a standard USB extender cable to get the UZB antenna further away from the PC.
In case the UZB is not loaded with a Z-Way license (the stick is generally sold in two versions, one with a license and one without for use with 3rd party software), the license can be loaded once Z-Way is up and running. Please use the **Z-Wave Expert User Interface** as described in Section 7 to apply the license. The license is a simple string that usually comes printed in a scratch card. Go to Network > Controller Info and click on the button License Upgrade. You will see a dialog as shown in Figure 2.5.
The Buy extension button leads you to instructions about how to extend the capabilities of the UZB stick by buying extra licensing files. The input field below allows inserting and applying the license key manually. Please note that:

- You must be connected to the internet to activate the license key.
- Every license key can only be used once (like scratch cards for prepaid phones).
- It is possible to apply various license files to the same hardware.
- The license is stored in the hardware. You can do a complete reinstallation of Z-Way on your platform or connect the UZB to a totally new platform without losing the license. However, loosing or damaging the UZB key means loosing the license!

It is possible to run multiple UZBs on one single hardware platform or even combine a RaZberry shield with a UZB on the same Raspberry Pi. Each piece of hardware will then manage its own network of Z-Wave devices having its own Home ID. However, Z-Way allows using devices of different Z-Wave networks together. This can be used to use products with different frequencies in one controller.
In order to enable a second Z-Wave transceiver (dedicated onboard, UZB or RaZberry), please use the standard user interface as described in Chapter 4. Go to the app management section and start another instance of the app **'Z-Wave**

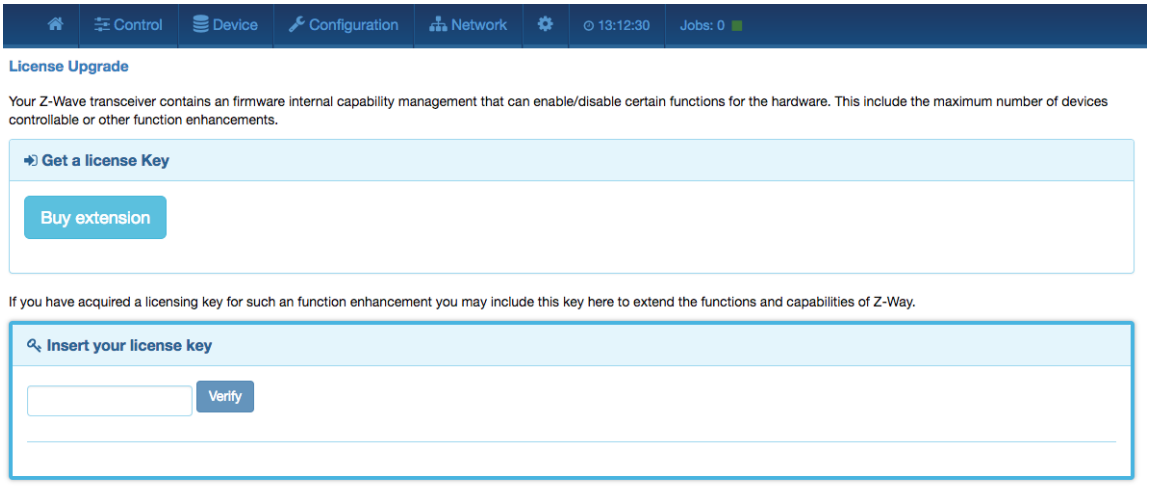


Figure 2.5: UZB license upgrade

Network Access'. Choose the new virtual serial device created by the new hardware.
Please note that the standard user interface will support devices from two networks, but you need to use the **Z-Wave Expert User Interface** to manage the second network. The inclusion and exclusion functions of the standard user interface will always use the first Z-Wave network.

2.2.1 Boot-Up Self-Test

On being powered up, the blue LED will light up, indicating that the self-testing has started. After about two seconds, the LED goes off, indicating that that self-testing has been done successfully. If the LED remains lit, it means that the self-testing has failed or the device is not booting up. You will need to replace the hardware in such a scenario.

2.2.2 Frequencies

The Z-Wave transceiver itself can be tuned into every frequency used by Z-Wave. However, to protect the transceiver and Z-Wave from high energy emissions on nearby frequencies (primarily 4G/LTE cellular radios using the 852 MHz frequency band), an external antenna filter is used. This limits the frequency changes to countries that share the same antenna filter. As of now, there are three antenna filter versions identified by their SKU codes.

- SKU: ZMEEUZB2 (865…869 MHz):
  - Europe (EU)[default]
  - India (IN)
  - Russia (RU)
  - PR China (CN)
  - RSA (EU)
  - Middle East(EU)
- SKU: ZMEUUZB2 (908 ... 917 MHz):
  - All the Americas except Brazil and Peru (US) [default]
  - Israel (ISL)
- SKU: ZMEAUZB2 (919 ... 921 MHz):
  - Australia/New Zealand/Brazil/Peru/Malaysia (ANZ) [default]
  - Hongkong (HK)
  - Japan/Taiwan (JP)
  - Korea (KR)

There are two options to change the UZB operating frequency:

1. If you use **Z-Wave Expert User Interface** , just choose the frontend on Network › Management as shown in the figure. 2.3. For more information about this **Z-Wave Expert User Interface** , please refer to Chapter 7.
2. There is a shell script available at

   `http://www.z-wave.me/fileadmin/download/changezwf.sh`

   Just execute the script with

   changezwf.sh [COM Port] [US|EU|ANZ|...]

2.2.3 Certifications

The UZB is certified for use in different countries.

CE / Europe

The UZB complies with the new Radio Equipment Directive of the European Union in general and the EN 300 220 version 3.1.1 in particular. Full CE declaration can be found in Annex **??**. The device also complies with the European ROHS and REACH regulations.

FCC / North America

The UZB stick shield was successfully tested for FCC. The FCC identifier is

2AAYUZMEUUZB.

Z-Wave Plus

The UZB USB stick is a certified hardware platform and a complete solution according to Z-Wave Plus. Please refer to the certification database

`http://products.z-wavealliance.org`

for more details.

2.3 Other hardware platforms

It is possible to port Z-Way to other hardware platforms beyond what is supported by binary distributions. Before contacting the Z-Wave.Me team, you can check if your platform meets the requirements for Z-Way to run on. The general requirements are:

- min. 200 MHz CPU clock speed,
- CPU architecture based on ARM, Intel or MIPS, as well as a GNU-based development tool chain,
- min. 16 MB Flash memory and 12 MB RAM,
- Operating system supports POSIX-compatible API.

There is a simple test to check if certain hardware on a platform is capable of running Z-Way. Follow the instructions given on

http://razberry.z-wave.me/index.php?id=28.

Only after you have double-checked that a binary distribution runs on your system or that the compatibility test has been passed, you may want to contact the Z-Wave.Me team for further discussions about porting and licensing fees.

3 Preparation and Ways to Access the System

3.1 Installation on Raspberry Pi

Before you can use the RaZberry solution, you need to complete your gateway hardware and install the software. Here are two ways to install and start Z-Way:

- A: You do not have Linux OS on your Raspberry Pi installed yet. Please download an SD card image (minimum 8 GB) from the download section of

  `http://razberry.z-wave.me`

  . It is based on the Raspberry Pi distribution "Jessie."
- B (recommended): You already have a working Linux (Jessie) running. Log in and execute the following command line:

  wget -q -O - https://storage.z-wave.me/RaspbianInstall | sudo bash

  .

You may need to configure the Wi-Fi access point of your Raspberry Pi in order to allow direct wireless access.

3.2 Installation on other platforms using UZB

3.2.1 Unix-based Platforms

Let's assume you already have shell access to your system. You can download a binary distribution from

`https://razberry.z-wave.me/z-way-server`

.

and unpack it. The code can run on every place in the filesystem. Nevertheless, we recommend using **/opt/z-way-server** as the folder to store the codebase. The folder content looks as shown in Figure 3.1. For more information on the files and the file structure of Z-Way, please refer to Chapter 11.4.

Z-Way can then be started using the simple shell command from inside the Z-Way folder:

LD_LIBRARY_PATH=./libs ./z-way-server

Once the code is started, it is possible to access the standard user interface using

`http://localhost:8083`

To use Z-Wave, the correct port of the Z-Wave transceivers virtual serial port must be configured. Please go to the user interface menu and open the app **'Z-Way Network Access'** as shown in Figure 3.2. for more information about Z-Way aps please refer to Chapter 6.

After checking the right virtual port created by the operating system, please configure this port name and save the settings. Now Z-Way should be up and running, and you may want to create a startup script to make sure Z-Way is running right after booting.

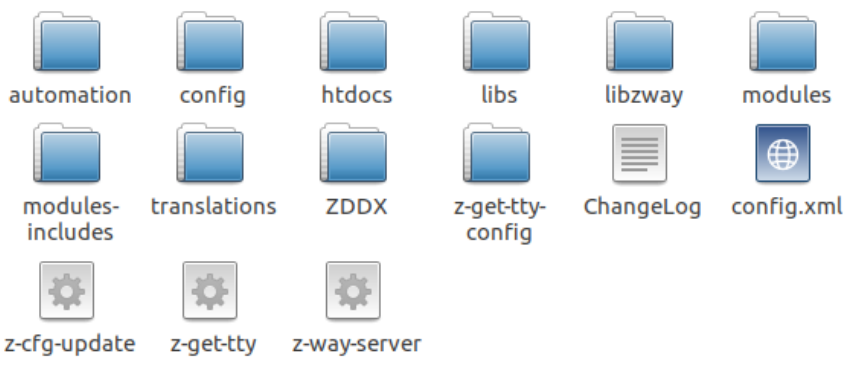


Figure 3.1: Folder Content of Z-Way

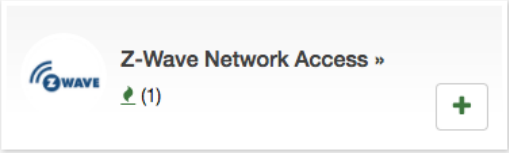


Figure 3.2: Z-Wave Network Access App

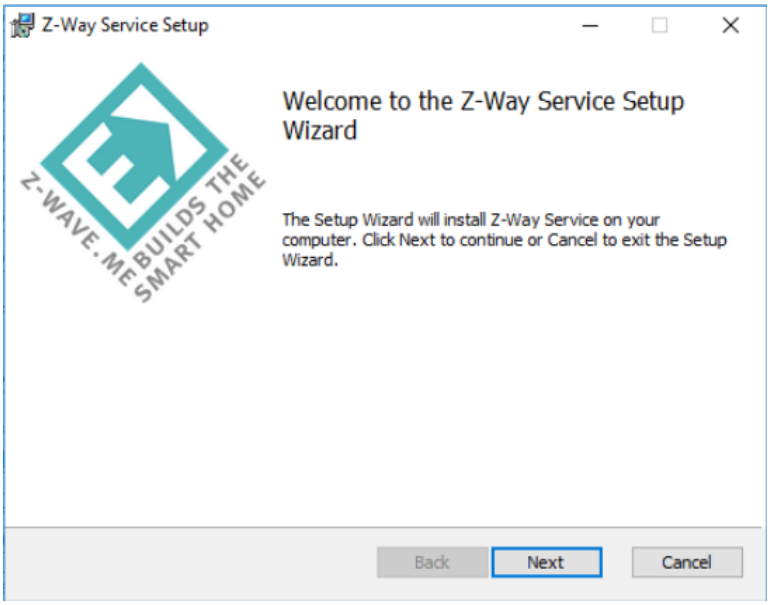


Figure 3.3: Z-Way Windows Setup Wizard

3.2.2 Windows

First, note that the Windows platform is only partly supported. There is a binary distribution for Windows, but it certainly lacks the level of testing required. Please use the binary Windows distribution at your own risk.
Install the MSI file and start it. You will see a nice wizard—see Figure 3.3 —guiding you through the setup process.
The files are installed on the folder of choice as shown in Figure 3.4.
If the UZB stick is plugged into the system, there is a new COM port created for this stick. Figure 3.5 shows the Windows hardware manager with the new port. After passing the initial setup page, open the app **'Z-Way Network Access'** and double check the right COM port. Note that there is a very special syntax for the COM port **'\\.\COM3'** for serial port 3.
Figure 3.6 shows this dialog.
Under Windows, Z-Way runs as a service. You find the service entry in the Windows Service Management as shown in Figure 3.7. This dialog also allows starting and stopping the service.

3.3 Local and Remote Access

Z-Way can be accessed in several ways:

- Using a standard web browser[1] on the controller's IP address. There is an embedded webserver on Port 8083 providing the web pages of the user interface.
- Using a standard web browser but the redirection service

  ```
  https://find.z-wave.me
  ```

  The user interface is similar, but there is no need to be on the same IP network or to install an explicit port forwarding.
- Use one of the native apps from Google Playstore or Apple iTunes. These will, however, use the two same services for local and remote access as mentioned above and will only render the user interface differently.

[1] We recommend Chrome, Firefox, or Safari since we frequently see problems with MSIE.

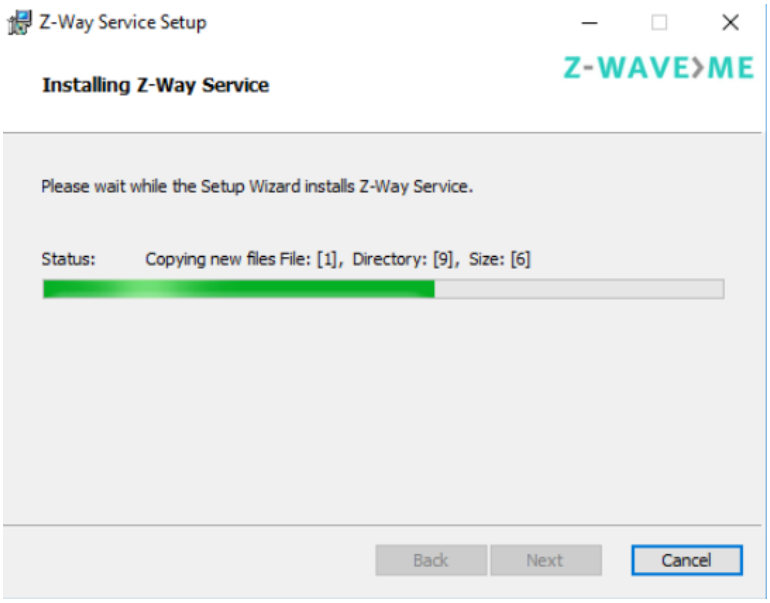


Figure 3.4: Z-Way Windows Installation

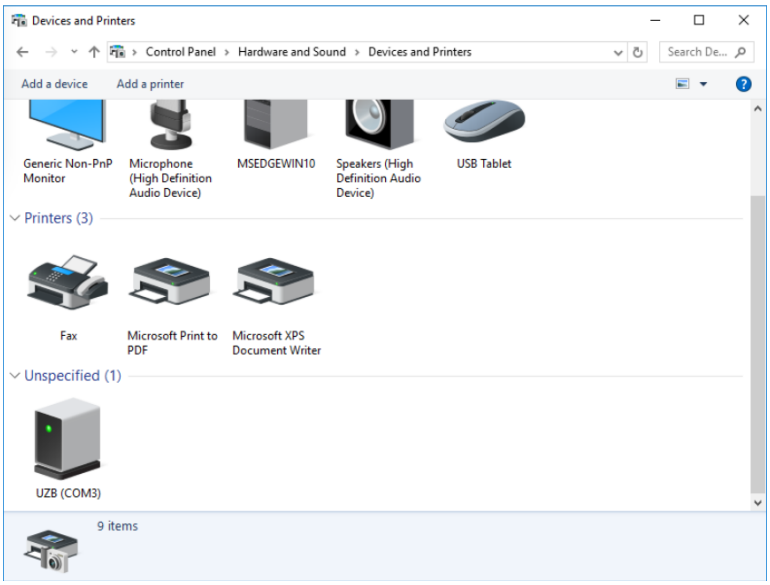


Figure 3.5: Windows Hardware manager with new COM port

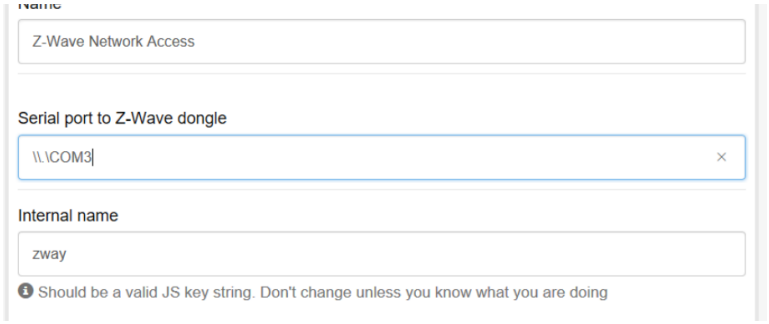


Figure 3.6: Z-Wave Network Access App with COM Port

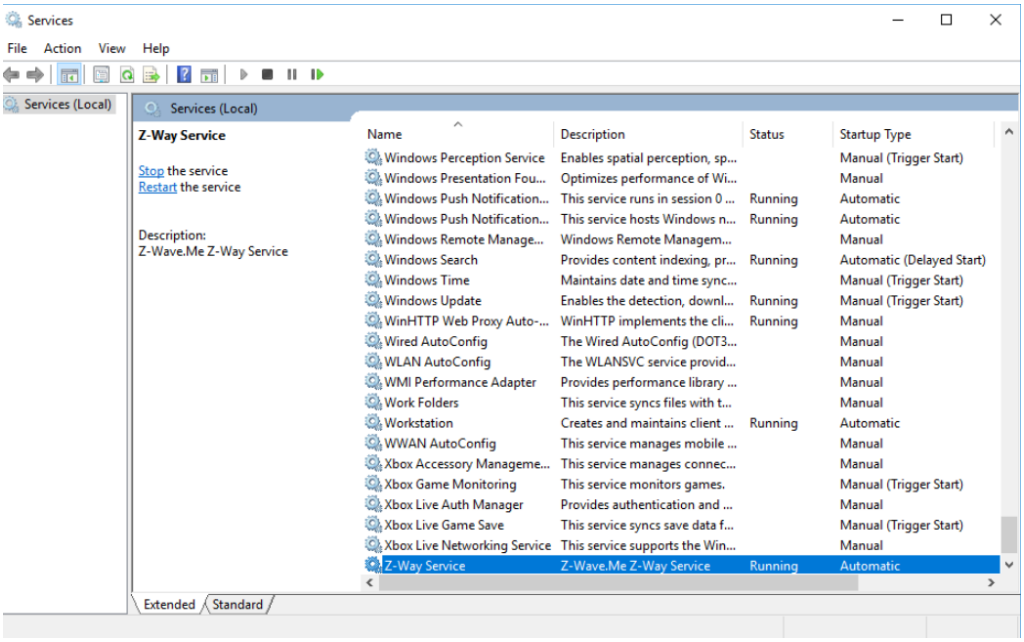


Figure 3.7: Z-Wave as Windows service

Access	Advantage	Disadvantage
Local IP Ethernet	very fast, all data stays within own home	no secure connection due to missing IP certificates, IP address must be known, no access from outside the home
Find Service	worked independent of local network setting from everywhere, very safe due to complete end-2-end security	depends on external service, more delays
Local IP WIFI	very fast and secure	another WIFI access point, no access from outside the home

Table 3.1: Comparison of Access methods

- Use your own web or native app-based user interface. Again, you will then use one of the two options mentioned above.

Both the local access and the remote access using the find-service have their pros and cons as listed in Table 3.1:
The initial access to the user interface must be done using one of the local access routes. Once Z-Way is running, the first access to the interface asks for some basic setup such as the admin password and an email address to recover this password. Figure 3.8 shows this screen. The number provided in the title (1) is the unique **Z-Way Platform ID** of the device and it will be needed in the future to access the controller remotely using the find service.

Please remember this **Z-Way Platform ID** !

The on the upper right side allows changing the interface language from standard English to your chosen language. If your language is not yet available, you may want to consider contributing a translation. Please refer to Chapter 10.3 for details on how to do this.
The language selection can be used for the user interface from this time on, but it can be changed in ☰ » My Settings.
Please note that the email address provided for email recovery will not be stored outside your controller, e.g. in a cloud service. This provides extra security but also means that certain processes like restore are a bit more complicated than is usual.
After completing the setup, a welcome screen will guide you through the system and introduce the basic user interface terms/dialogs.

- Element,
- Element View,

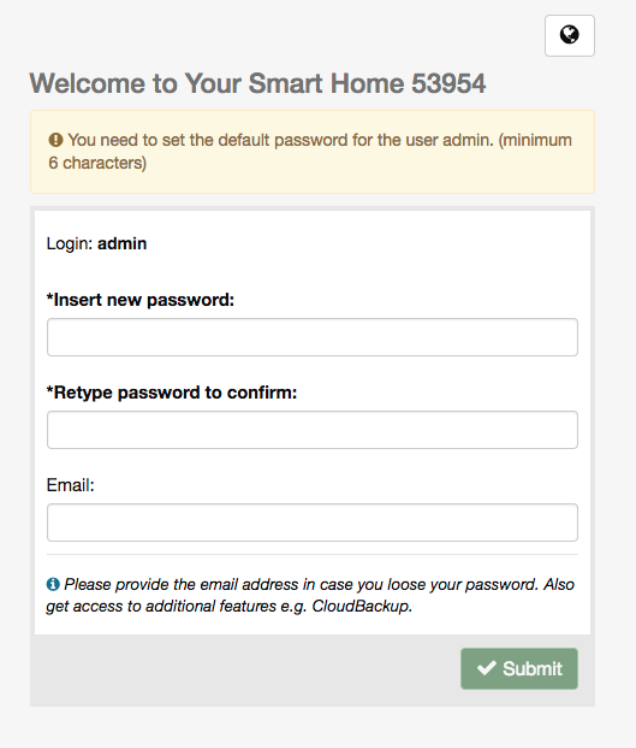


Figure 3.8: Initial setup of the Z-Way User Interface

- Element Configuration,
- Dashboard,
- Event,
- Timeline,
- Apps

It also offers buttons for direct access to the two most common actions right after installation:

- Add a new physical device.
- Add some virtual device, internet service or application.

After completing the setup, it is possible to access Z-Way using the find service, or install a native app for the mobile phone. Open the url

`https://find.z-wave.me/`

The dialog on this find service, as shown in Figure 3.10, is intentionally simple. It offers two ways to log in:

- Using your **Z-Way Platform ID** , along with your login name (Example: '23333/admin') and your password for remote login and redirection
- In case you use the service from a PC or a mobile phone within your home, the service will detect this and show the IP addresses of your Z-Way controllers. It is possible that a Z-Way controller has two IP interfaces (Ethernet + Wi-Fi). In this case, both addresses are shown. Clicking on the IP address will lead to the local IP login screen, as shown in Figure 3.9, where only name (e.g. admin) and password are required.

The menu option Logout in the setup menu of the user interface will terminate the session and return you to the login screen or the find service depending on how to log in. You can always force the termination of the find session by calling the URL

`https://find.z-wave.me/zboxweb`

.

3.4 Security and Privacy

Security and privacy are of great importance. Z-Way tries to maximize security and privacy and will not compromise them in order to improve user experience and convenience.

- All user data including login name and email address are stored only locally, which means that the controller must be available and connected to restore passwords. We believe that this is a robust security and privacy measure.

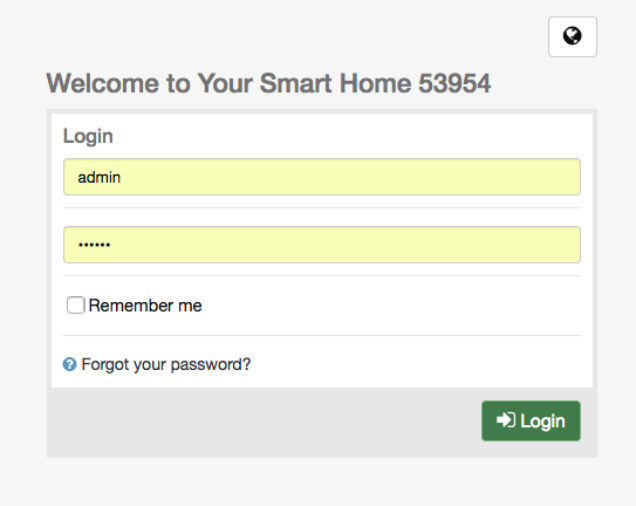


Figure 3.9: Login on local IP address

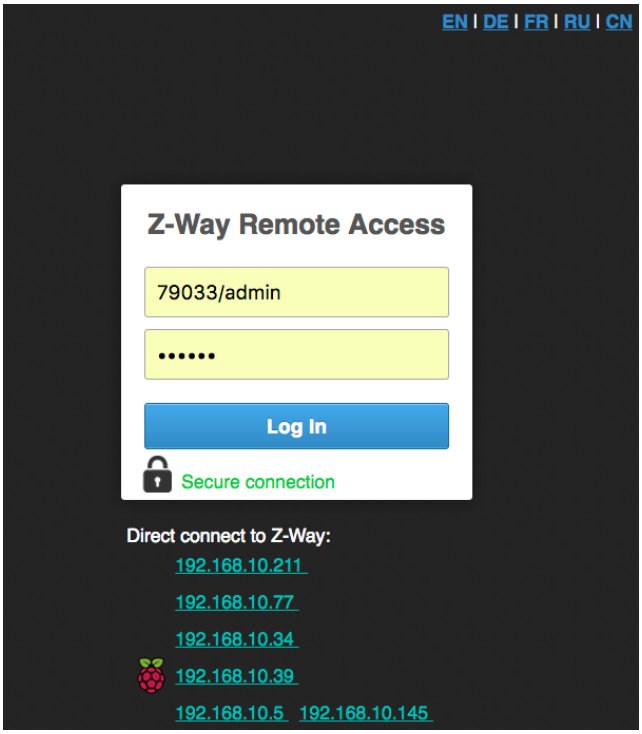


Figure 3.10: Remote Login Screen

- Z-Way offers certain cloud-based services. This requires certain services and connections to the Z-Way cloud service:
  - Right after boot-up, the Z-Way controller will connect to the find service announcing its presence. This is done using a reverse ssh service. The client side of this service is available on the Z-Way server and you can review its work and source code. If you don't like this service, you can turn it off using the Z-Way Smart Home Interface . However, be prepared not to have the find service available anymore.
  - The backup to cloud service will also create a copy of your device data on the cloud service. This is very convenient and ensures that you have a backup file when needed. If you feel uncomfortable leaving a copy of your smart home configuration on the Z-Way cloud service, just don't use this service and turn it off.
- The find service allows https connection and provides a valid certificate issued by COMODO RSA Domain Validation Secure Server CA. Please check the validity of the certificate before using the find service. This means that the access using the find service has established a complete secure connection from the web browser via the find service to your controller at home.
- There is no https access enabled to the local port. This is because there is no way to create a valid certificate on an IP address that is assigned dynamically using DHCP. It would still be possible to run encryption with https, but we believe that this would be mimicking security without having real security. That's why we only keep http to send a clear message about the risk of accessing a local IP address. We strongly recommend doing an initial password setting and subsequent local access using Wi-Fi only since Wi-Fi comes with its own very secure encryption based on WPA.

4 The Web Browser User Interface

The Z-Way Smart Home Interface is considered the main control interface of Z-Way.
A screenshot overview is shown in Figure 4.1 marking the essential parts of the interface. It looks similar on different devices such as a desktop PC, smartphone, and tablet (both native app and browser), but will adapt to the screen size.
The Z-Way Smart Home Interface follows some very clear logic thats was developed by Prof. Christian Paetz and presented the first time on the Smart Home Summit 2014 in Munich/Germany.
The following basic rules apply:

1. **Elements**: Every function of any device is shown as one single (No. 7). (In case a physical device has multiple functions, like switching and metering, it will be offered as multiple elements). All elements are listed in the elements view (No. 3) and can be filtered by function type (switch, dimmer, sensor) or other filtering criteria.
2. **Elements Configuration**: Every element offers a configuration interface (No. 8) for changing names, removing it from the screens, etc. Important elements can be placed in the Dashboard (No. 1).
3. **Event**: Every change in a sensor value or a switching status is called an "Event" and is shown in the timeline (No. 4). Filtering allows monitoring the changes in one single function or device. Besides, elements can be assigned to different rooms (No. 2).
4. **Apps**: All other functions such as time-triggered actions, the use of information from the internet, scenes plugin of other technologies, and services are realized in apps accessible in the setup menu (No 6). These apps are ready-to-use scripts/templates that can add extra logic and functionality such as logic rules like "IF->THEN," scene definitions, timers, and interactions with external (non-Z-Wave) devices connected via USB dongle or via internet. Some apps are built into the system. More can be downloaded from an app store. To use an app, you create an instance of this app and configure its properties. If useful, you can create more than one instance of one single app. The apps can create none, one, or multiple new elements and events. You can install new apps and manage them using the menu Configuration -> Apps.

4.1 Z-Way Smart Home Interface Daily Usage

4.1.1 Standard Element View

Figure 4.1 shows the standard element view of the Z-Way Smart Home Interface . Elements with an icon, buttons, or variables are placed side by side. Depending on the screen size, they are grouped in one (mobile phone), two (pad), or three (PC screen) columns.
The very same display is available on the Dashboard (No. 1), the Room View (No. 2), and the Element view (No. 3). The element view shows all elements (No. 7) available while the dashboard shows elements that where manually selected for display on the dashboard. The clock on the topline menu (No. 5) shows the actual time in the time zone of the gateway (not the actual position of the browser). The elements can be filtered by element type or can be ordered by

- Time of creation (ascending or descending)
- Given name in alphabetical order (ascending or descending)
- Last update
- Custom

The button Drag-and-Drop turns the user interface in a reordering mode. Just drag and drop elements to reorder them. After the new order is saved, it is available as 'custom'. The same logic can be applied to all view showing elements (Element View, Dashboard, and Room View).
A search field allows searching for given names of the elements.
All elements can be assigned to "tags": Tags are text blocks that can be arbitrarily chosen. Typical tags are "Temperatures," "Energy," and "Outdoor." Multiple elements can have the same tag and one element can have several tags. By selecting a tag, only elements are shown that are tagged with this name.
Tags are managed on the element configurations described below.
Figure 4.2 shows two typical elements: Each element has a given name. This name may refer to the function or the position. Please make sure to keep the name short enough so that no display problem is created. Each element also has one or several icons. In case the element represents an actor function, the icon usually refers to the switching state of the actor (on or off, up or down). Changing icons also indicate the status of binary sensors such as motions detectors. Analog sensors typically only have one icon. The upper element is an actor allowing to be switched on and off.

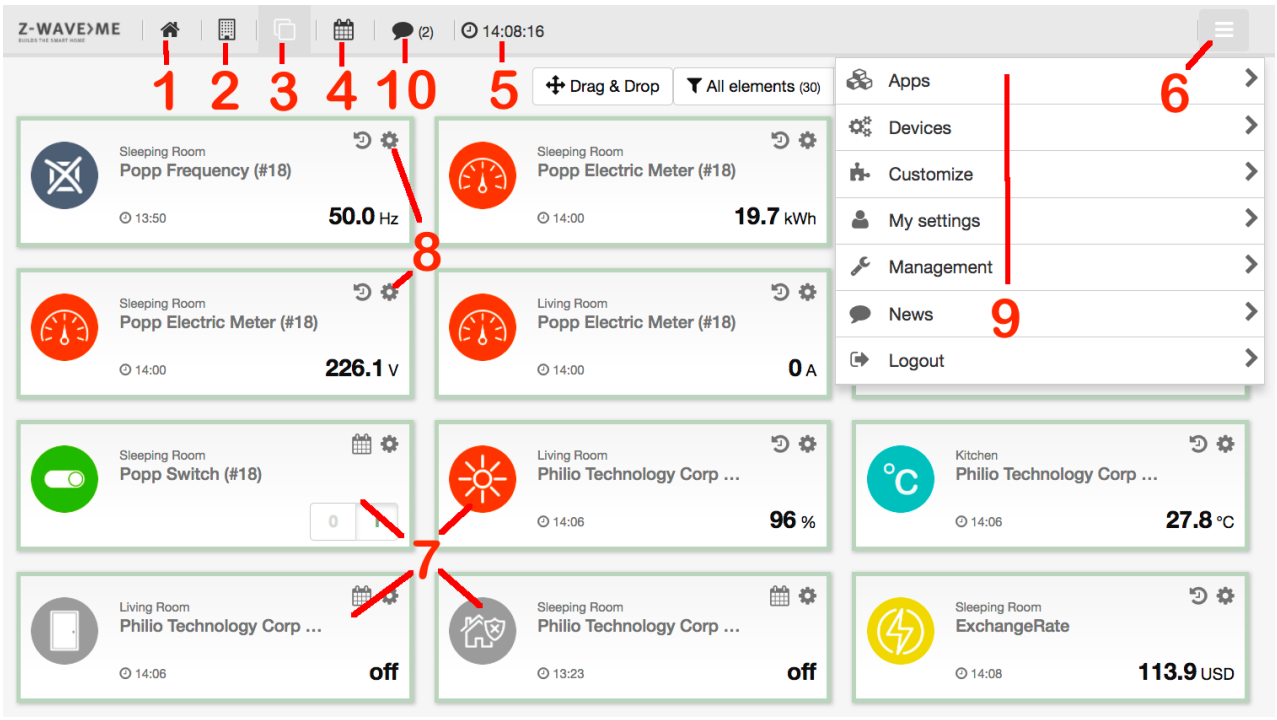


Figure 4.1: Z-Way Smart Home Interface

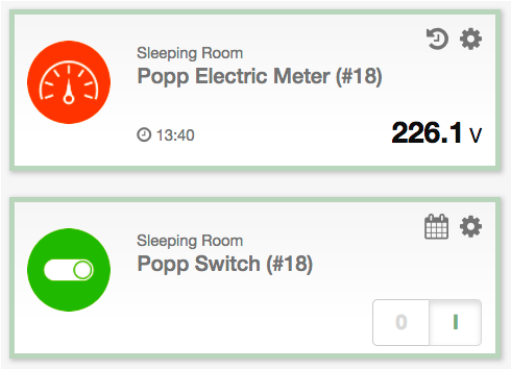


Figure 4.2: Elements

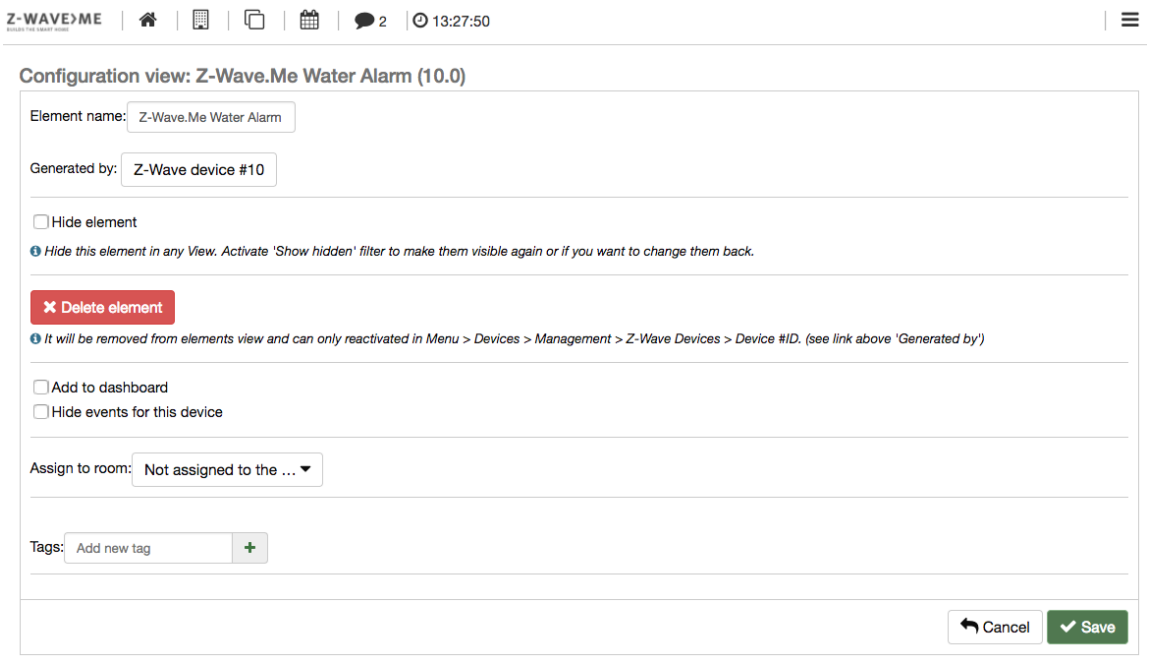


Figure 4.3: Elements configuration - upper part

The (No. 8 on Figure 4.1) on the right-hand side opens the element configuration dialog. The symbol left to the configuration wheel depends on the type of device. For devices with analog values such as rain sensors, temperatures, etc., the icon will open a 24-hour history of the sensor value. Devices with 24-hour history also show a time stamp when the last value update was received. Clicking on the large icon itself will call for a value update, but please keep in mind that battery-operated devices will only send updated values after the next wakeup. For event-driven devices such as actuators or binary sensors, an icon will show a list of the last 10 events with the time stamp. One click away is then the full list of events, as described in Section 4.1.1, but filtered for this element.

Element Configuration

Every element shown has its own configuration dialog (No. 7 on 4.1). Clicking on the symbol on the upper right-hand side opens this dialog.
Figure 4.3 shows the upper part of the element configuration dialog.
The element name is the given name of the element. Z-Way tries to automatically find a reasonable name, but the user should change this name according to the specific setup of the home.

- Generated By: This refers to the physical device of the Z-Way application that generated this element. In the example above, this device is the physical Z-Wave device with Node ID 2. Clicking on the button with the name of the element-creator leads to the configuration dialog of the physical device, i.e. the Z-Way application.
- Hide element: As explained in the dialog, this checkbox will hide the element. However, this setting can be reversed by showing hidden devices. This setting is for cosmetic view only.
- Delete Element: This checkbox allows removing the element. It will not only disappear from the element view but also from any dropdown list to setup device relationships, etc. For information on how to re-activate such an element, refer to the user settings dialog description in Section 4.2.4.
- Add to Dashboard: This places the element on the dashboard.
- Hide events from this device: This keeps the device in the element view or Dashboard, but no events of this device will be shown in the event timeline.
- Assign to room: This allows assigning this element to a certain room or changing this assignment.

Figure 4.4 shows the lower part of the element configuration dialog. The tag option allows setting and removing tags. When a tag name is inserted into the text field, it autocompletes known tag names.
The custom icon sub-dialog allows changing the icons of the element. Each element has one, two, or three icons depending on the status indicated. Each of the icons can be replaced by an own individual icon. Just click on the pencil button to change the icon. To download more icons, refer to the customization section in the configuration

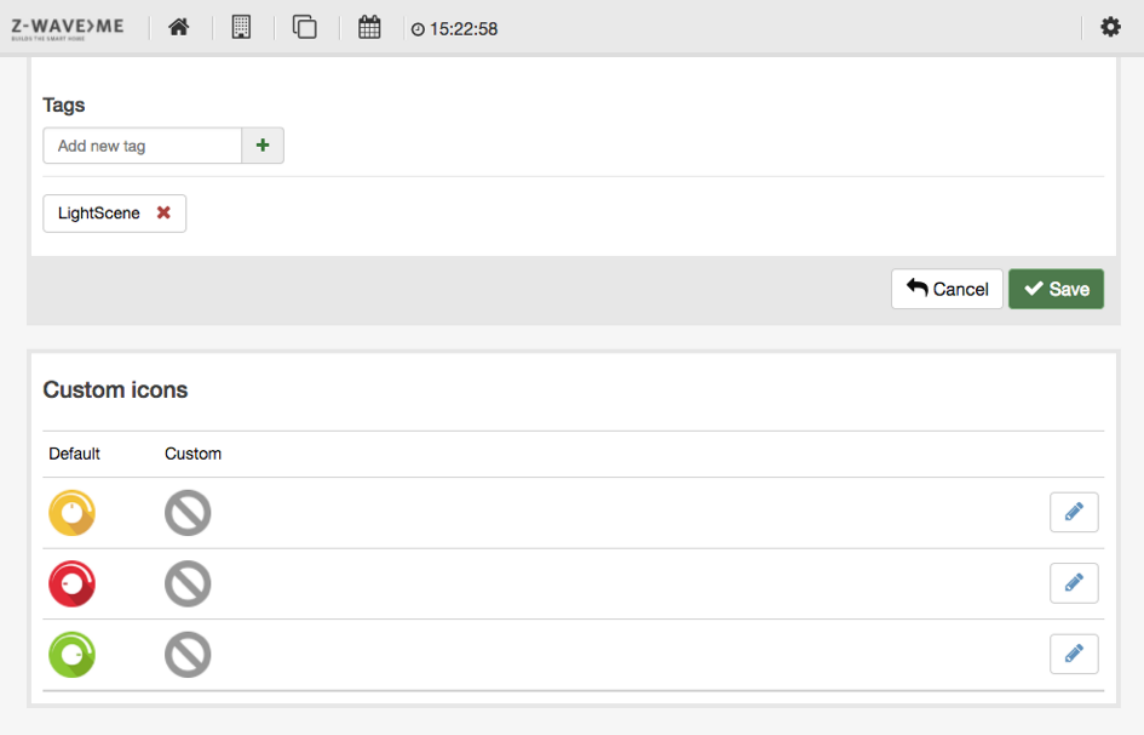


Figure 4.4: Elements configuration - lower part

menu.

Room View

The Z-Way Smart Home Interface allows managing different rooms and assigning elements to rooms. Each element can be placed in one room only.
Clicking on the room symbol on the top menu opens the room view with a list of rooms, as shown in Figure 4.5.
Each room has its own element with a background image that can be configured. It is possible to add new rooms using the button labeled Add room. The room is named and the number of elements in this room is shown. Per room there are up to three sensors that can be selected as "quick-view sensors". They are shown right below the name and in the top menu in the individual room view.
Clicking on the room element leads to the room view, as shown in Figure 4.6.
The room view lists all the elements in the room. As in the element view, they can be re-arranged using the drag-and-drop feature.

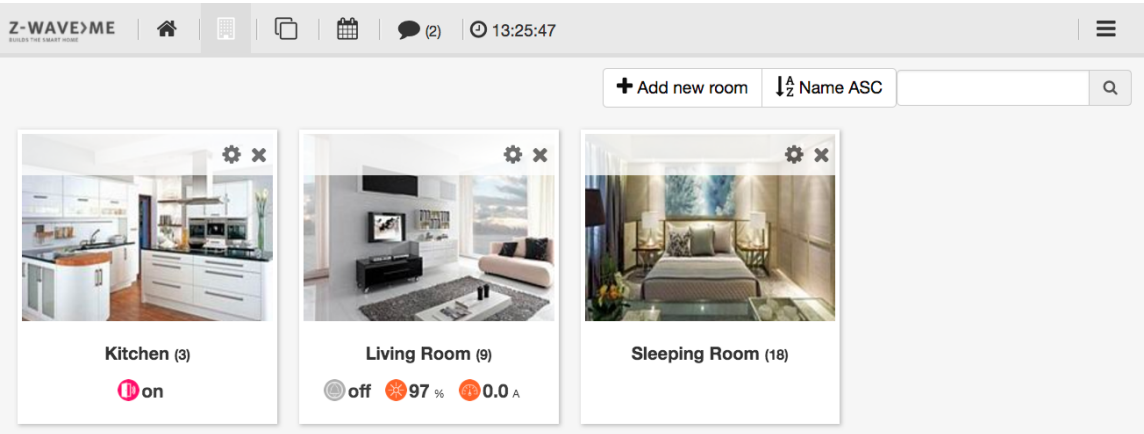


Figure 4.5: Room overview

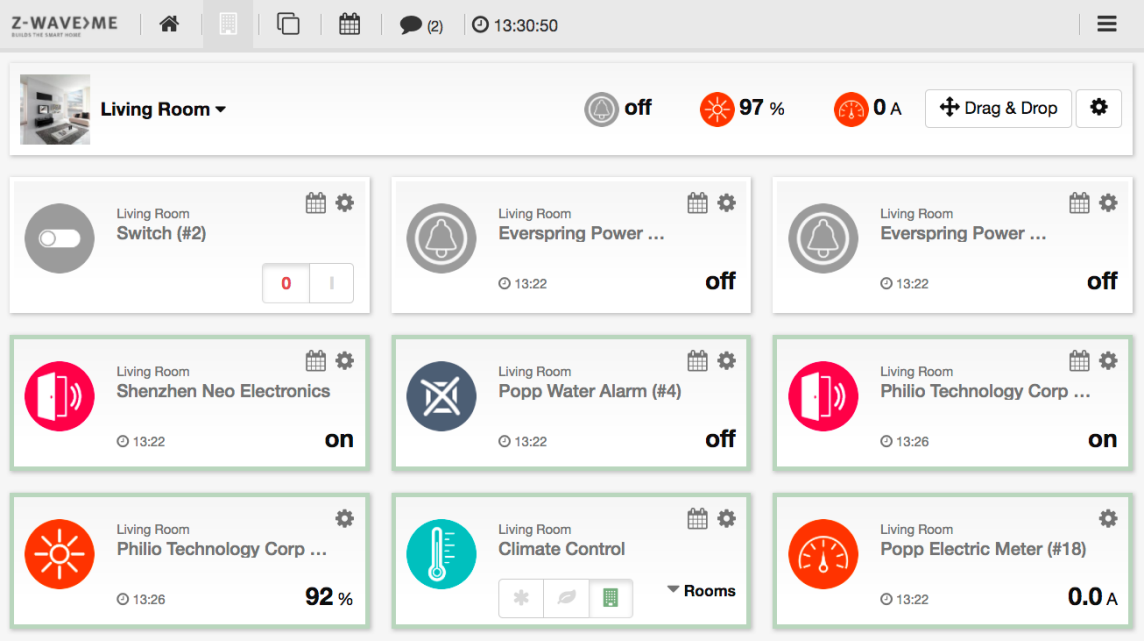


Figure 4.6: Room View

The top line shows up to three quick sensors and allows quick change of the room using the dropdown list. If your browser supports gestures, you can change the rooms by swiping left or right.

Clicking on the ⚙ symbol of the room element or inside the room view in the top line opens the room configuration dialog, as shown in Figure 4.7.
Each room can have an individual name. Besides some pre-installed images, it is possible to upload images for the room. A checkbox defines the room image that is used as a background image in the room view.
The dialog allows assigning elements to this room which were not assigned to any room yet. Just click on the element name in the list of Available devices.
The little checkbox on the right-hand side allows selecting up to three sensors as “quick-view sensors”.

Event Timeline

The event timeline is the forth menu item in the top menu.
The timeline dialog offers a chronological view of the events in the Smart Home. The events are:

- Change of status of actuators such as dimmers, blind controls, or switches
- Tripping of binary sensors (motion, door, etc.)
- Change of measured value of sensors
- Network status changes (device lost, device back, etc.)

The standard view of the timeline, as shown in Figure 4.8, lists all events with the icon of the element, room info, time stamp, name of the element, and status info. Every line item has a context menu on the right-hand side. This menu allows

- showing events of this source element only,
- showing events of this event type only,
- showing events that have the same event type,
- directly moving to the element configuration page of this element,
- hiding all events from this source.

4.1.2 News feed

Each Z-Way controller is connected to an RSS feed. This feed contains news and alerts about the platform. Whenever there is a new feed entry, the top menu bar will indicate this with a sign, as shown with Marker 1 in Figure 4.9. Clicking on this opens the full list of news. This full list can also be accessed using the ’News’ menu item in the configuration section, as described in Section 4.2.5.

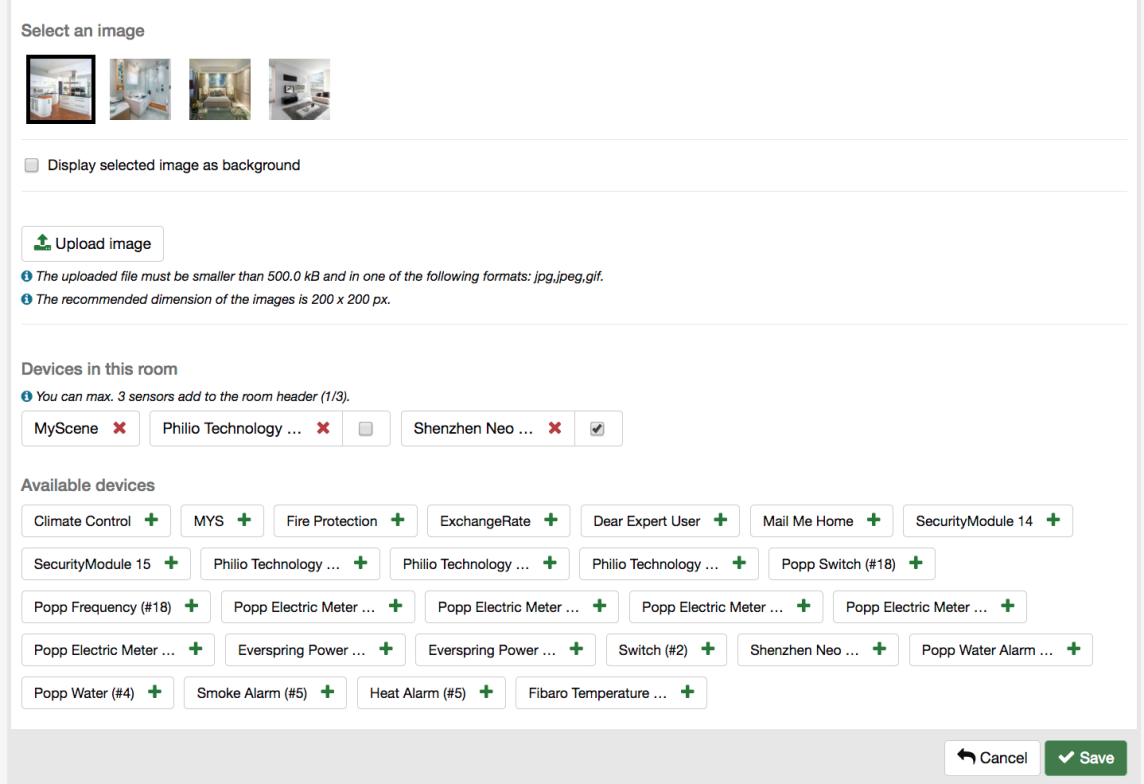


Figure 4.7: Room configuration dialog

Figure 4.8: Timeline

Figure 4.9: News Indicator

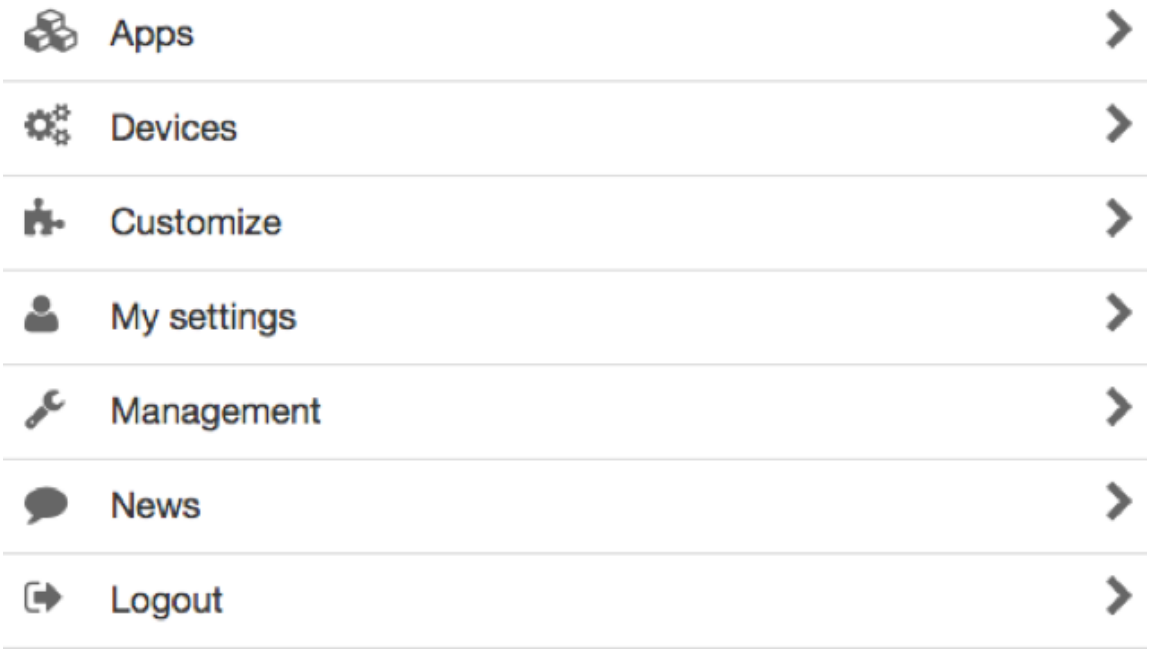


Figure 4.10: Configuration menu

4.2 The Configuration Menu

Clicking on the menu item (marked as No. 2 in Figure 4.9) on the right-hand side opens the configuration menu. The configuration offers various functions to enhance and configure the smart home system as such and the user interface. Figure 4.10 shows the configuration menu of Z-Way.

4.2.1 Apps

This menu option allows managing the home automation applications and interfaces of Internet or IP-based services or devices.

Z-Way apps are like software applications that use the infrastructure of the Z-Wave network to provide application solutions and dependencies. These software applications also extend the capabilities of the network and implement automation functions.

Like any other application software such as those for PCs, some Z-Way apps are preinstalled on the device, and others can be downloaded and installed by the user. Like application software, some software solutions can only run once while others can be started multiple times.

The app menu has three parts:

- List of apps locally available for use.
- List of apps available on the central server and ready for download.
- List of apps that are active and running.

The list of local apps, as displayed in Figure 4.11, shows a small subset of apps that are already on the local devices. The top part shows some of the apps that are most frequently used (featured apps). A filter allows filtering for certain app types; ordering and direct search in the search box also help to find the right app. If there are active instances of the app (software running x times with different parameters), this is indicated right below the name of the app.

Clicking on the name will open a dialog with further information about the app. Clicking on the + button will lead right to the configuration page of the app.

The online app list, as shown in Figure 4.12, offers the same functions. However, apps need to be downloaded first before they can be configured and started. Clicking on the download button will copy the app to the local repository and start the configuration dialog, as shown in Figure 4.13.

Each app has its own name that can be changed. Depending on the function of the app, there are several different setup parameters.

Please note again that some apps can be started multiple times while other apps are "singletons." They must only run once on the system. Once they run, they will disappear from the repository since they cannot be started again. Configuration of such a singleton is still possible using the menu of running apps. If such an

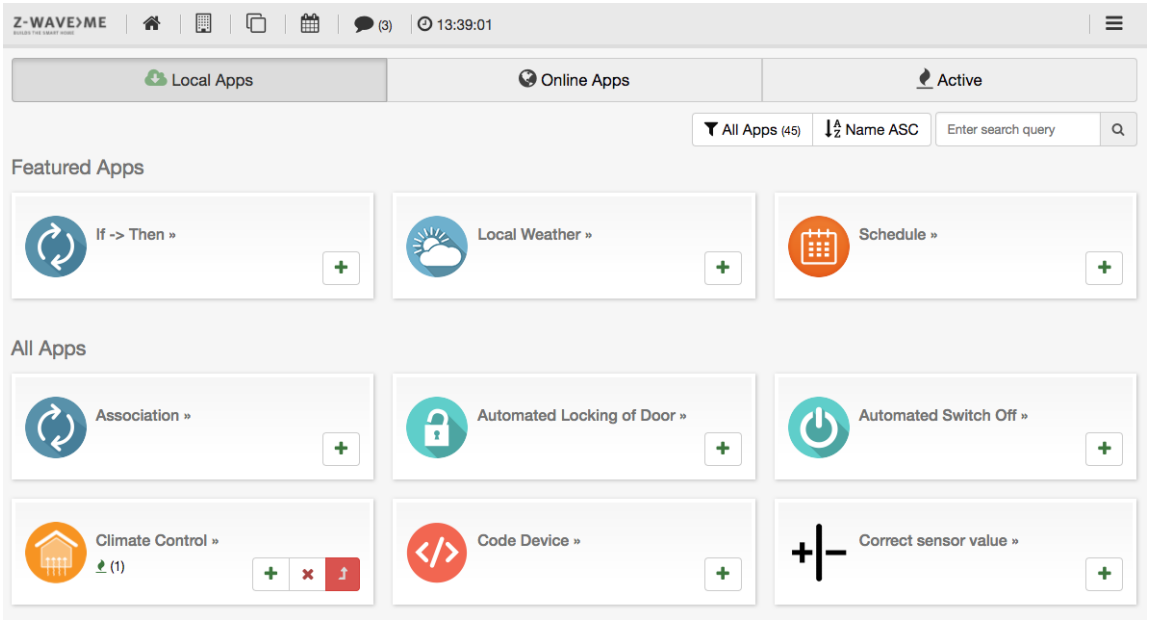


Figure 4.11: Local Apps

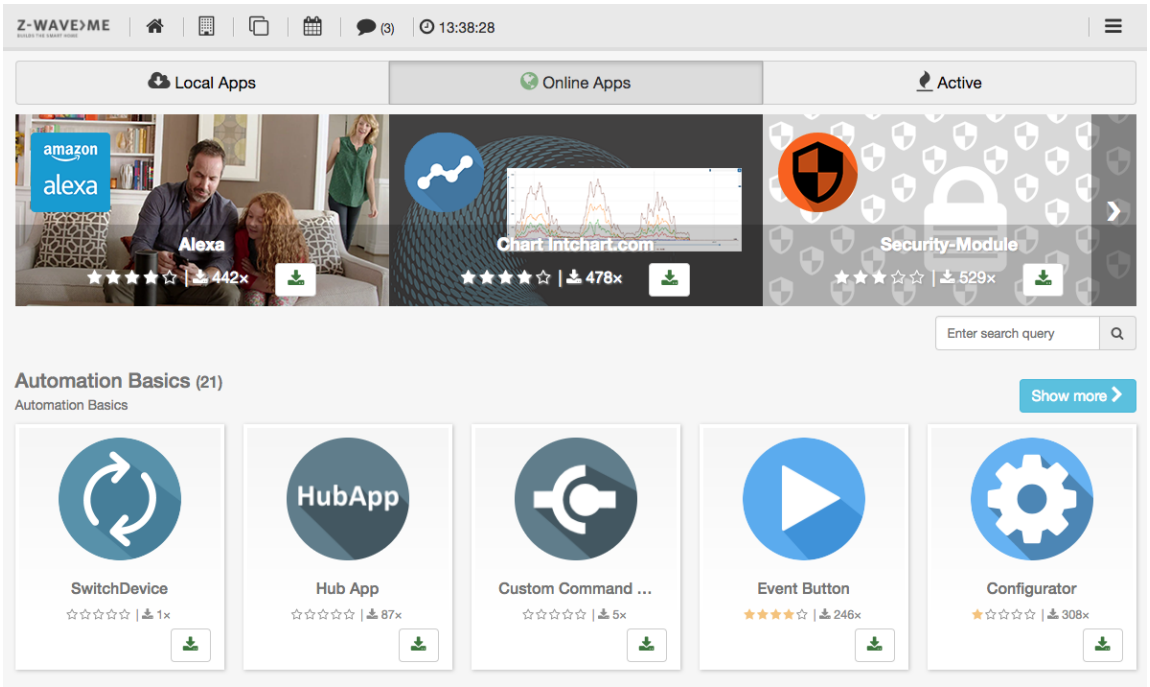


Figure 4.12: Online Apps

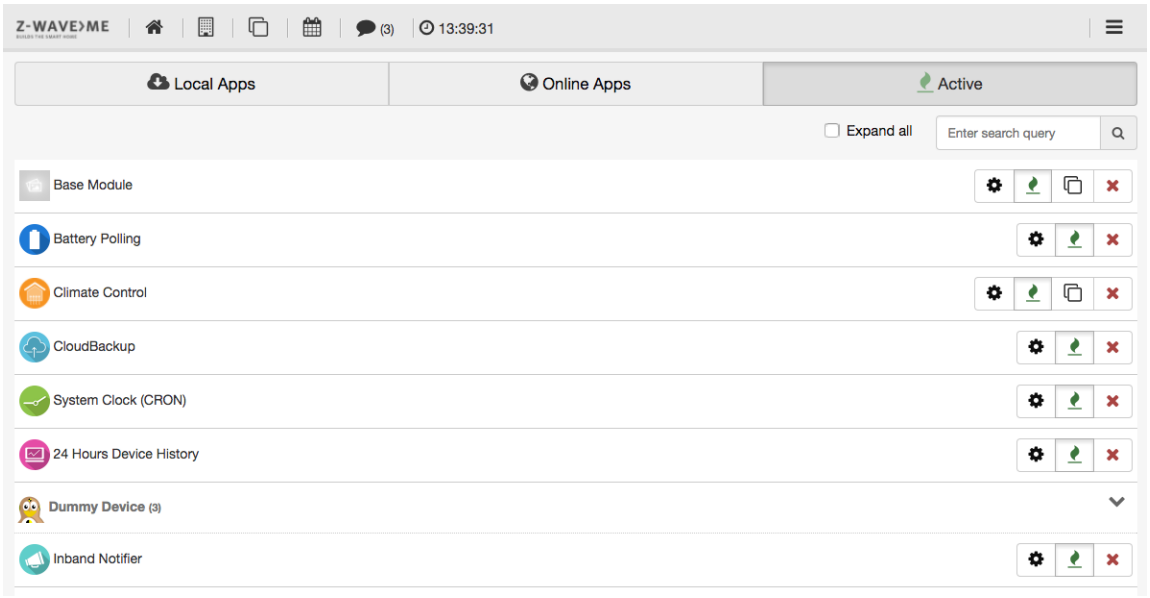


Figure 4.13: App Setup

app stops, the app entry will reappear in the local repository.
The tab for active app management is shown in Figure 4.14. It lists all active apps by app type. If there are more than one app of the same type (e.g. **'IF->THEN'** is usually needed multiple times), the view is collapsed for this type of application but can be expanded. It is possible to expand all sections of multiple app types using the checkbox on the top. It is also possible to search for certain app names.
Every app line has a submenu with the following functions:

- : Stop this app
- : Allows cloning an app. This will open a dialog for a new instance of the same app with the same settings. After saving these settings, the new app becomes active and is shown in the list too.
- : Stop the App. It remains configured but is inactive. Once inactive, the green flame changes into a red ON button to restart the app.
- : Opens the configuration dialog. This is the very dialog to be completed during the initial start of the app.

Please refer to Chapter 6 for more information on different apps.

4.2.2 Devices

The device menu allows managing physical devices. By default, it offers two physical device types: "Z-Wave devices" and "IP cameras." However, if other wireless communication technologies are activated, they will be shown as well. Please refer to Chapter 9 for more information on how to integrate further wireless technologies. This section will also explain how to include/exclude and manage these devices and IP cameras.
Figure 4.15 shows the device list including EnOcean and 433 MHz devices.
Therefore, let us focus here on managing Z-Wave devices. Besides the standard buttons to add and manage devices of the specific communication technology, Z-Wave offers one more button to link to a specific second Z-Wave user interface for installers and professionals. Please refer to Chapter 7 for a detailed description of this very technical, the so-called **Z-Wave Expert User Interface** . Please note that all day-to-day management functions can be done without involving this very specific and technical interface.
In case the controller hardware supports the new Smart Start feature of Z-Wave, there will be another button to include new devices - the QR code san as sown in Figure 4.16.
Smart Start is a new way to include devices into Z-Wave using the QR code provided with S2 authentication. The user scans the QR code thats is stored in an internal so called provisioning list. Smart Start Devices will then announce to be included when powered up. In case the S2 key is in the provisioning list the controller will automatically include this new device without any further user interaction.
The button 'QR Code' opens a dialog to capture the Device Key, either by typing them in or by scanning the QR. Another menu tab allows managing the Device Keys already captured by not used for inclusion.

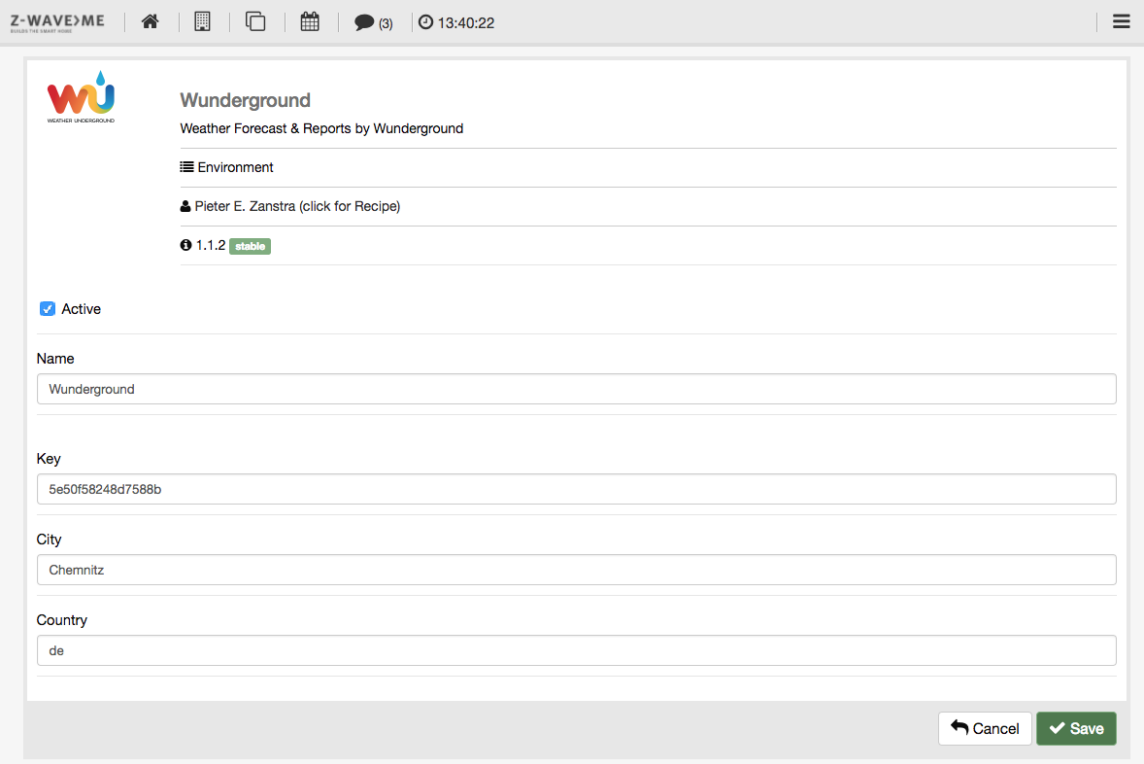


Figure 4.14: Active App Management

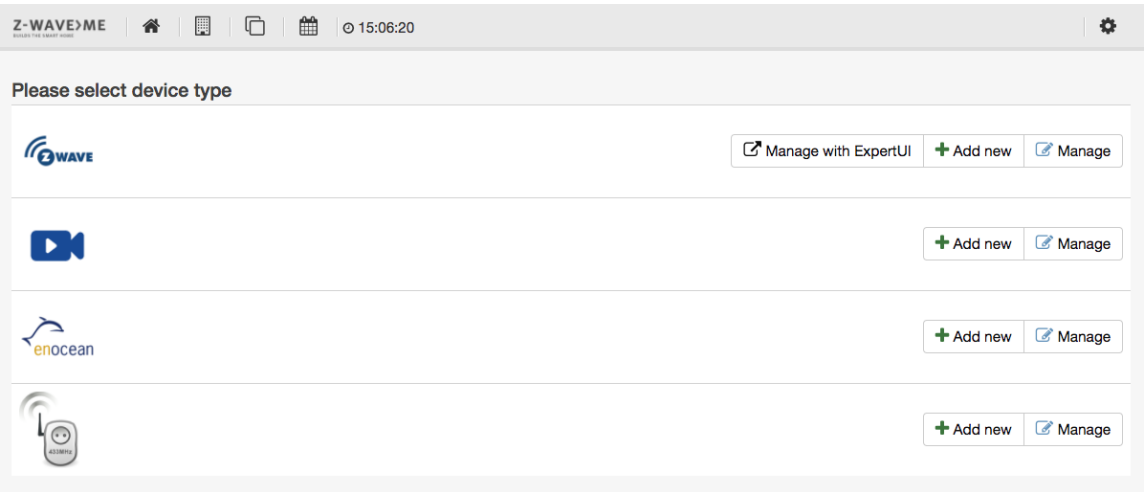


Figure 4.15: Device Management Overview

Figure 4.16: Scan QR Code for Smart Start

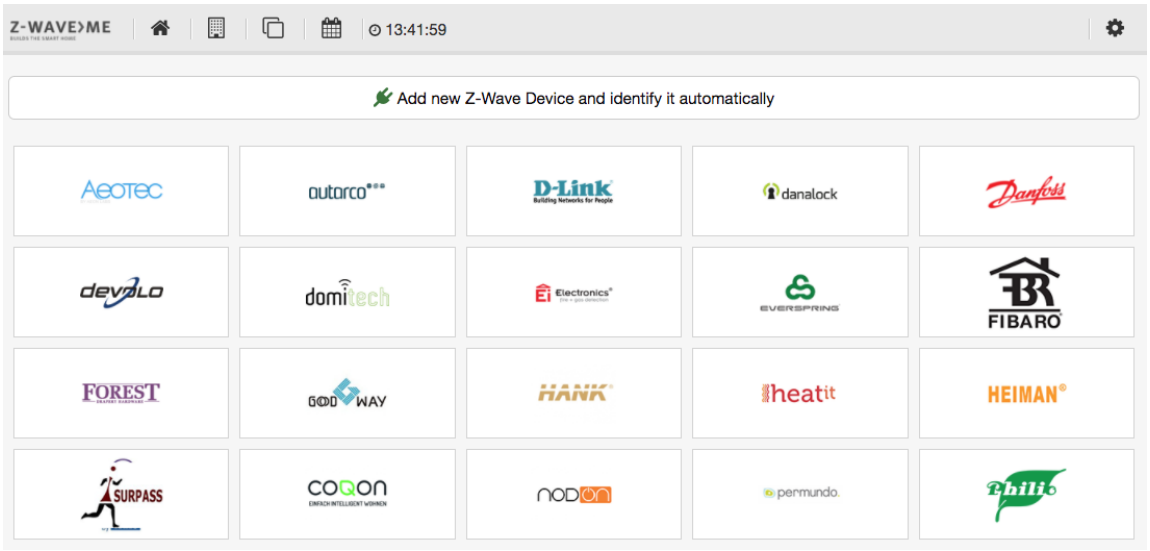


Figure 4.17: Z-Wave Device Vendor Overview

Please note that a standard web browser running on a standard PC may not provide the capability to scan QR codes.

Inclusion

To include a new Z-Wave device (one of the first steps needed to start a smart home system!) please click on the + button on the Z-Wave device part. This will open an overview of current Z-Wave device vendors by name and logo. Figure 4.17 shows this overview.

Generally, there is no need to know the Z-Wave brand and product code. All Z-Wave devices are self-describing, and automatically identified products will provide the same functions as the devices that were pre-identified. Thus, experienced users will always click on the upper button to add a new unidentified Z-Wave device. The only reason to find a specific device from the list is to get some additional information on how to include this device. This refers to the button and the button push sequence needed for inclusion.

Since most Z-Wave devices have one Z-Wave inclusion button and single or triple click will do the inclusion, this information is only needed for some devices with exotic inclusion options. Both the buttons to include an unknown device and the right-hand side button of an identified button will lead to the same inclusion dialog as shown in Figure 4.18.

It is recommended to exclude (reset) a device before it gets included.

However, if you are sure that the device is new and in factory default state, you may skip this step. Right next to the inclusion button there is another small button that defines if the device will be included with special security functions. By default, the security option is enforced. However, some devices in the market may not work as expected using the security function. In case there is a connection problem, unsecure inclusion may still work.

Once the inclusion mode has been started, the controller waits for devices to be included. Figure 4.19 shows the controller at this moment. The inclusion mode can be terminated using the same button. Any new device included will also terminate the process.

In case the new device requires authorization, this needs to be done right after inclusion. Authorization ensures that the device that appears on the user interface is indeed the device in hand. To ensure this the device offers either a QR code to be read or a device individual PIN number, both types of information need to be provided to the user interface manually. The controller will then match the information provided by the user with the information provided by the device using wireless communication. Only in case they match the device can be used.

Authentication is only required for certain devices. In this case, a window like that shown in Figure 4.21 pops up asking for the authentication information. Once given, they are checked. If authentication fails, a warning is displayed. It is not possible to just repeat the authentication. The device must be excluded and re-included.

Any new device will be interviewed next. In this process, all functions announced by the device itself will be verified

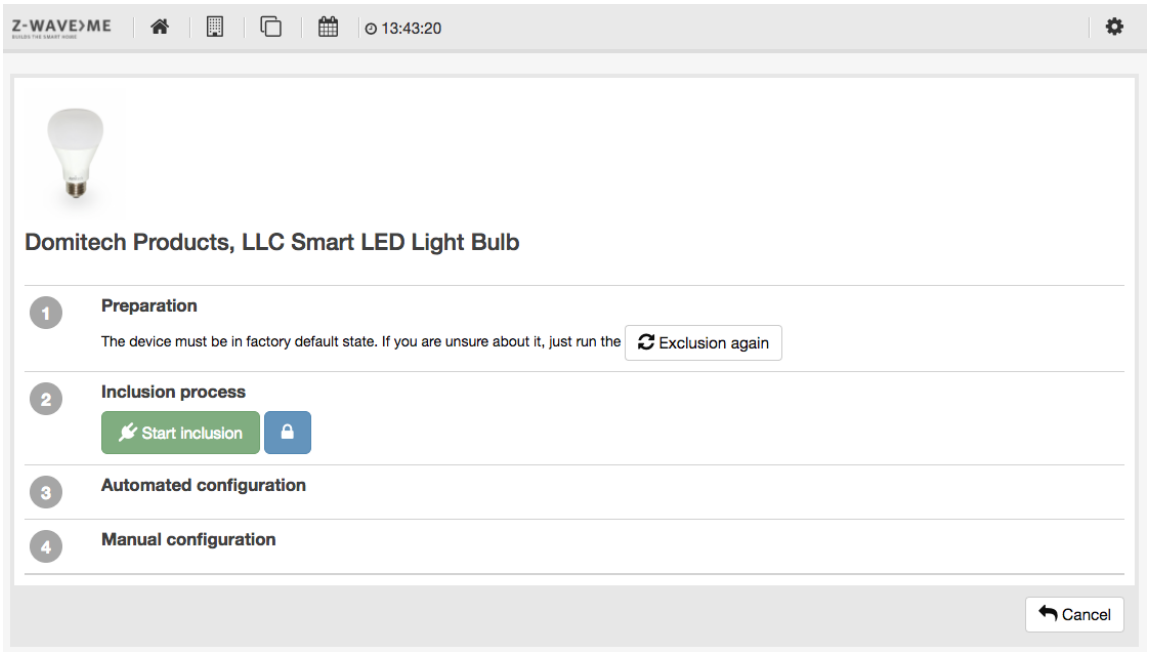


Figure 4.18: Z-Wave Device Inclusion Dialog

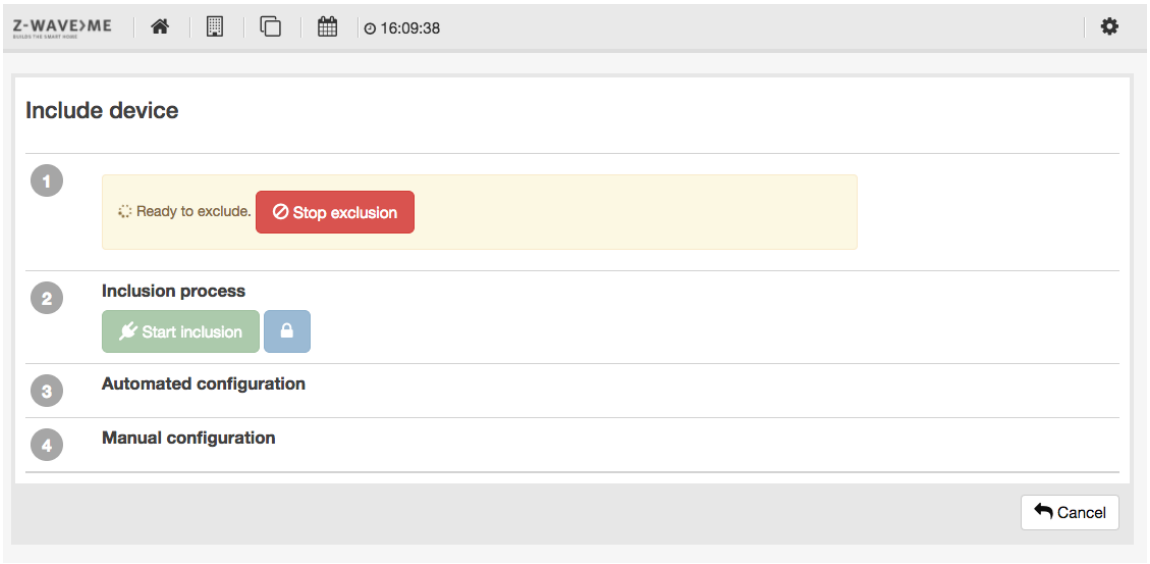


Figure 4.19: Z-Wave Device Exclusion Dialog

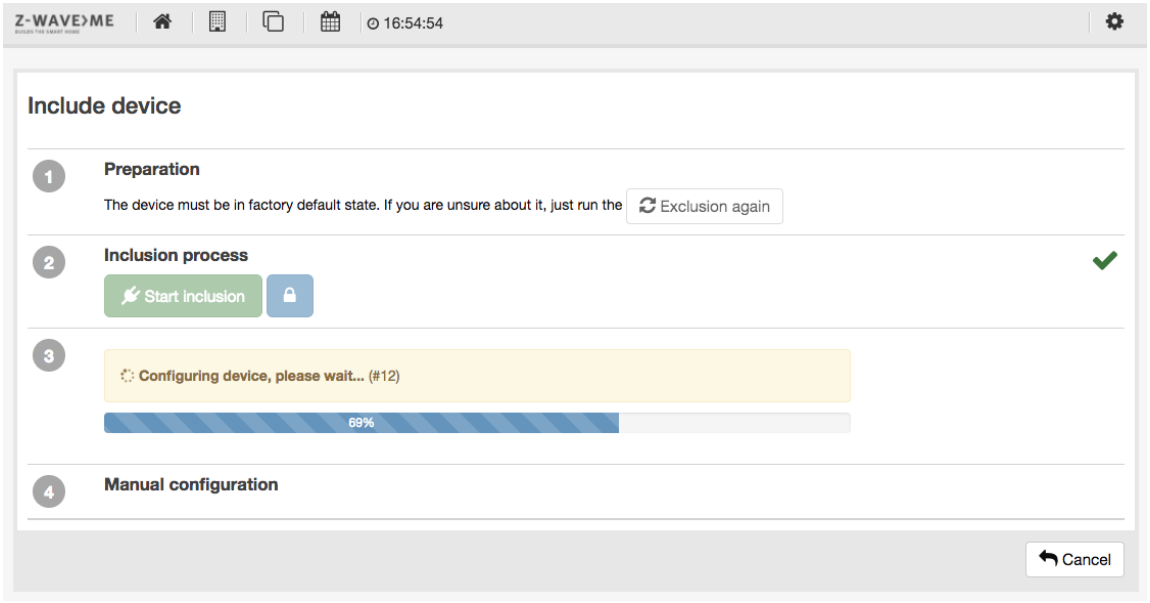


Figure 4.20: Z-Wave Device Successful Inclusion

Figure 4.21: Z-Wave Device Authentication

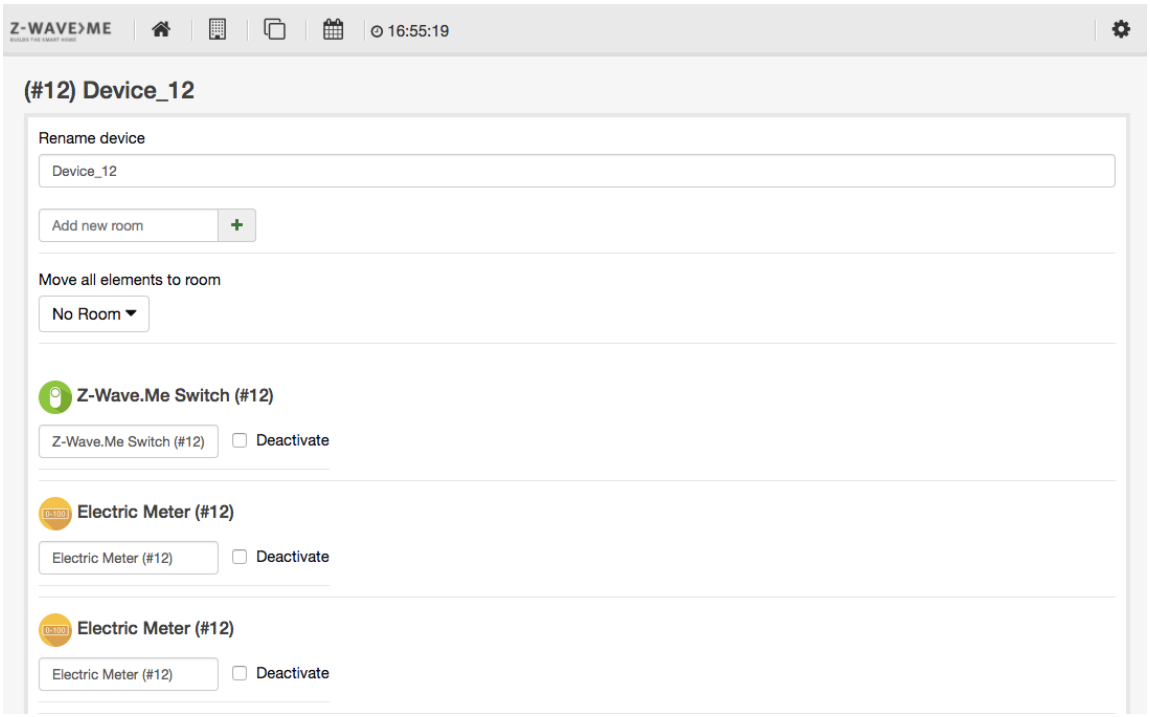


Figure 4.22: Z-Wave Device manual configuration

and certain user interface relevant data will be called from the device. A progress bar such as that in Figure 4.20 shows the status.
Once the interview was passed successfully, a dialog offers some initial manual configuration functions, as shown in Figure 4.18:

- Rename the device as such. This device name only refers to the physical device and will not be shown in the standard user interface. Use descriptive words like "Popp Smoke on sealing" to re-identify the device later.
- It is possible to move all elements shown below into one single room. If this is not done here, it is still possible to move each element on the configuration dialog as described in Section 4.1.1.
- The list of the elements generated by the new device. Here you can change the name that will then appear on the element overview, etc. You can also deactivate the element if you don't see any need to have it.
- Some physical devices offer further hardware-specific settings such as wakeup interval time. If the new device offers such configuration, another button for hardware configuration is shown. Please refer to the **Z-Wave Expert User Interface** configuration description in Chapter 7.4 for more information on how to use this dialog. Both dialogs are identical.

The interview process does not only detect all information from the device; it also tests the connectivity of the device. Certain communication may fail. Another good reason for such a failure is that a battery-operated device goes into deep sleep mode too fast. Figure 4.23 shows the error message in case of failure. In most cases, it's OK to just redo the interview and wake up the device.
If the second attempt at the interview fails, the controllers gives the option to accept the result or to redo the entire process. Figure 4.24 shows this dialog box.
Once the interview has passed and all configurations are done, the device can be used.

Z-Wave Device Management

The second option for Z-Wave devices besides adding (including) new devices is the device management menu. Clicking on the button Manage button opens a menu with three tabs:

- Figure 4.25 shows a list of the physical Z-Wave devices. The button will open the very same device management dialog as described during manual post-inclusion configuration in Section 4.2.2. The button opens a dialog to remove the device. As shown in Figure 4.28 there are two options:
  - Reset and Remove: This button will start the normal Z-Wave Exclusion Process. Exclusion requires that the device to be excluded is still functioning and accessable.
  - Remove: Only in case the device is defect nor not existent anymore this option shall be used. It will use the Z-Wave function 'Remove Failed Node' without any involvement of the device to be removed.

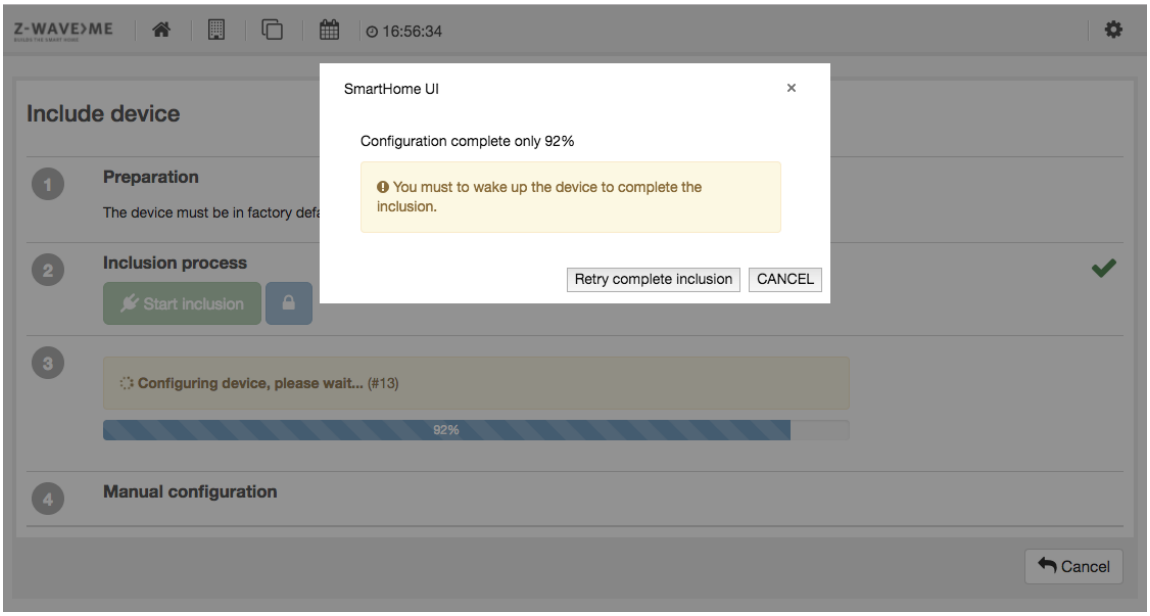


Figure 4.23: Z-Wave device inclusion failed

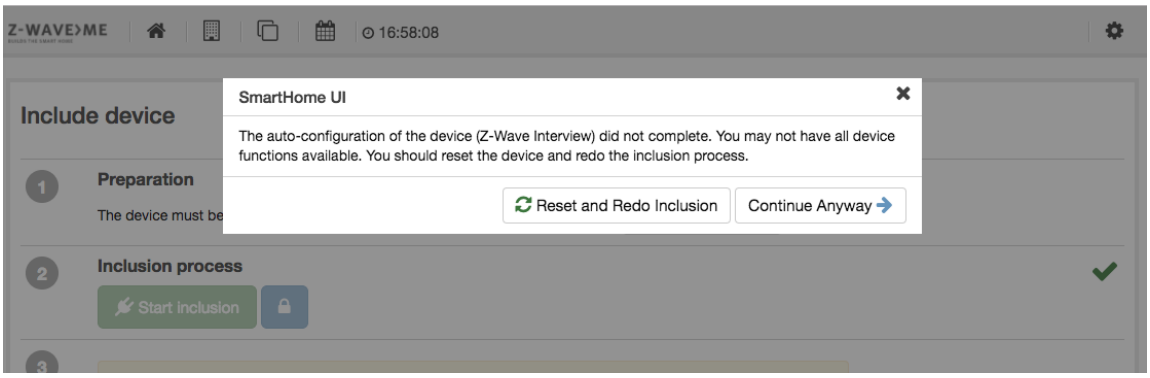


Figure 4.24: Z-Wave device inclusion repeated

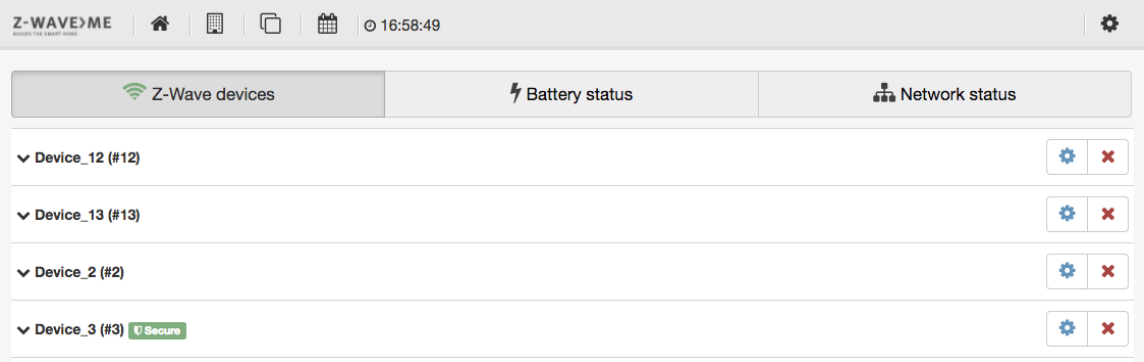


Figure 4.25: Z-Wave device overview

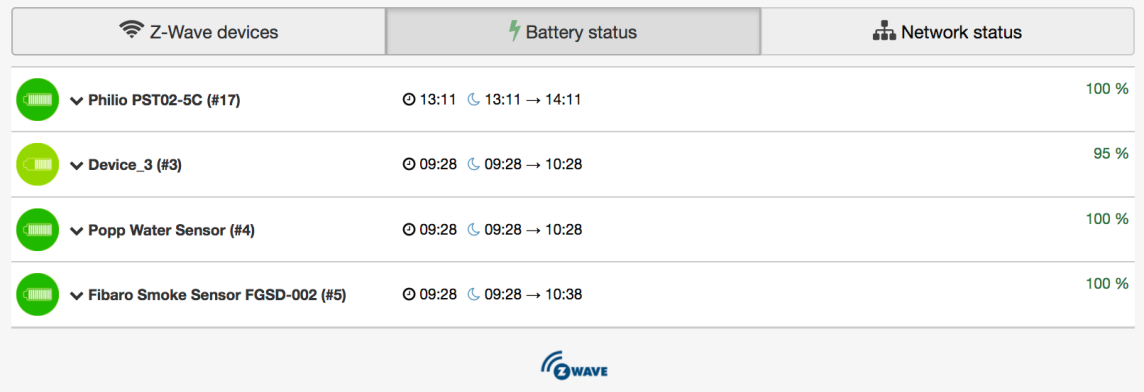


Figure 4.26: Z-Wave device battery overview

- Figure 4.26 shows the battery status overview. The list can be ordered by the battery charging level.
- Figure 4.27 shows a list of network status messages. This can be warnings for empty battery, devices lost, devices replaced, etc. Clicking on the device name lists all the elements created by this physical device.

4.2.3 Customize

The customization menu option allows changing the look and feel of your Z-Way user interface. You can add more icons as device-specific icons and you can change the look and feel using skins. For more information on skins and how to create them, please refer to Chapter 10.1. This menu here only deals with skins that are already available.

The menu offers four tabs. The tab shown in Figure 4.29 shows the list of skins locally available. They can be activated by just clicking on the green activation button.

The tab shown in Figure 4.30 offers the list of skins available for download. They must be downloaded first before they can be activated and applied.

The tab shown in Figure 4.31 lists the additional icons available on the controller. They can be activated per element on the element configuration menu described in Chapter 4.1.1.

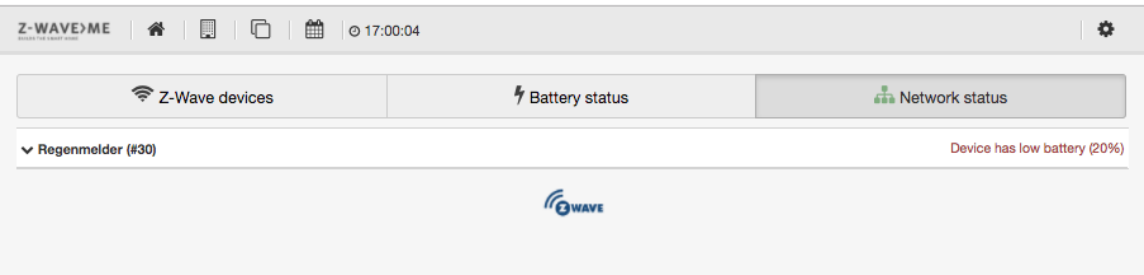


Figure 4.27: Z-Wave device network status

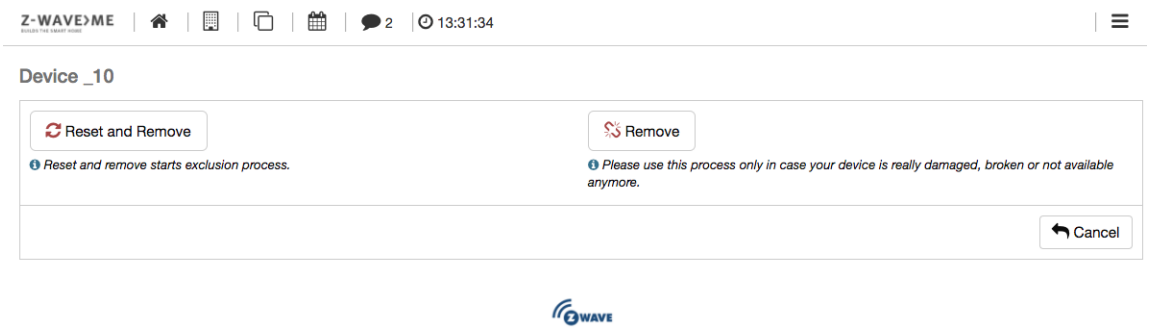


Figure 4.28: Z-Wave Device Reset /Exclusion

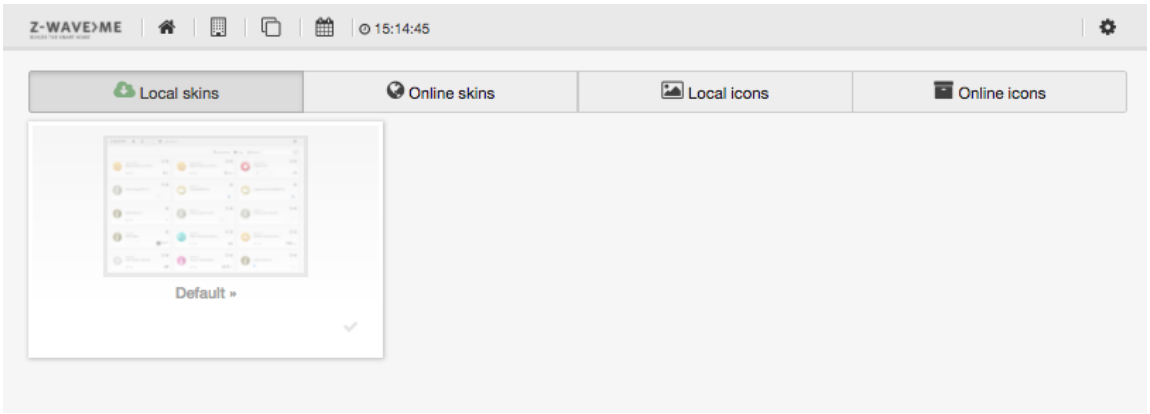


Figure 4.29: Local Skins

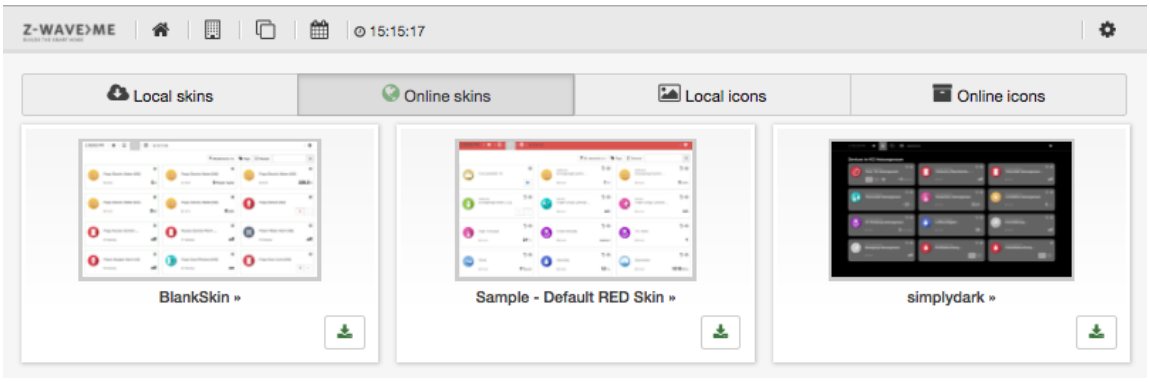


Figure 4.30: Skins on Server

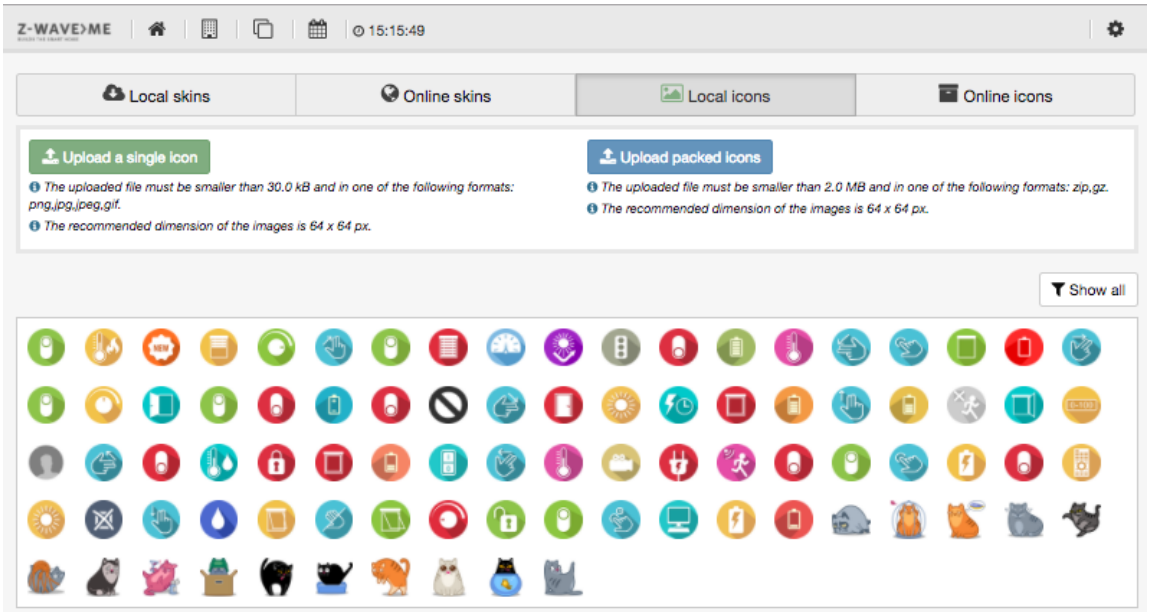


Figure 4.31: Local Icons

The tab shown in Figure 4.32 offers additional icon sets available on the server. They must be downloaded first before they can be activated and applied.

4.2.4 My Settings

In this dialog, as shown in Figure 4.33, the local user interface settings for the logged in user can be changed.

- Name: This is the name of the account. Even if this name is changed, the login name is NOT changed.
- Email: This email is used for certain email notifications, for password recovery and for the recovery of cloud backups.
- Language: Click on the flags to change the user interface language.
- UI update rate: This is the refresh rate for the user interface web pages.
- Expert View: Having the checkbox marked shows some system apps in the overview of running apps that are not shown by default. For more information on apps, please refer to Chapter 6.
- Events: The two checkboxes allow suppressing certain events. They are then removed from the timeline and will not create any out-of-band alert.
- Hidden events of devices: This is a list of the devices where events are deactivated in the element configuration overview. This is the only way to reactivate them if needed. For more information about the element configuration, dialog please refer to Chapter 4.1.1.

Figure 4.34 shows the lower part of the Settings dialog. In My User Account the individual password can be changed. The section Add mobile device shows a QR code to simplify the setup. Please refer to Section 5 for more information on how to use this QR code.

4.2.5 Management

The menu item Management opens a new menu with options to technically manage the platform. These management options are available for administrators only. Please refer to Section 4.3 for more information about the management options.

News

As shown in Figure 4.35, this dialog lists all news entries received from the RSS feed.

Logout

The logout button cancels the user sessions of the Z-Way Smart Home Interface . In case of a login from

```
https://find.z-wave.me
```

the user is redirected to the find.z-wave.me overview page or else to the local login page.

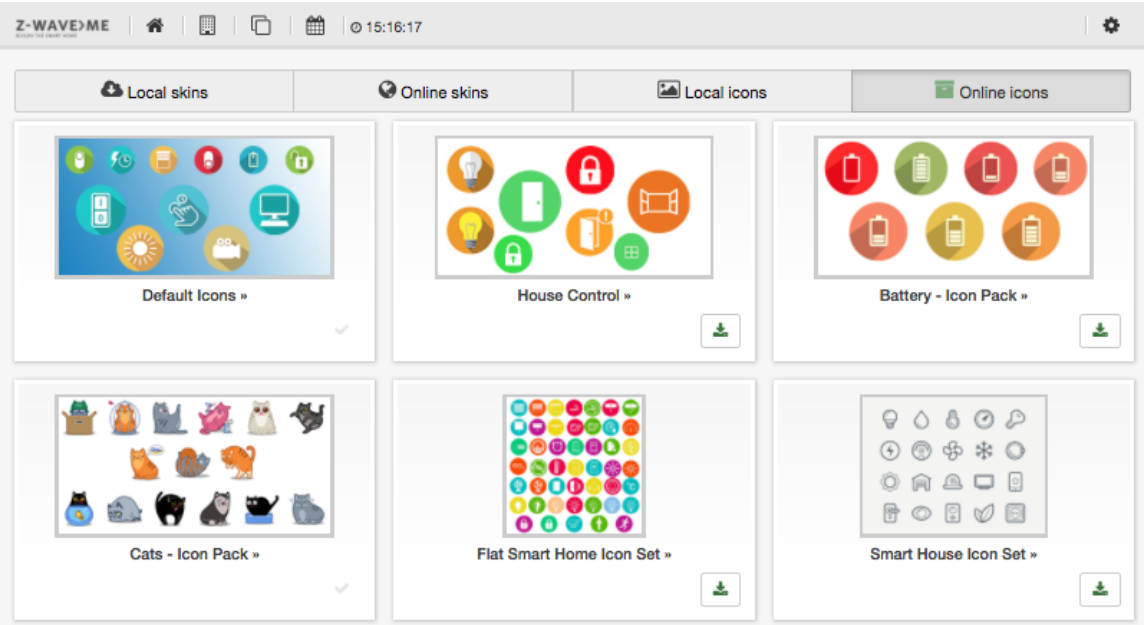


Figure 4.32: Icon-Sets on Server

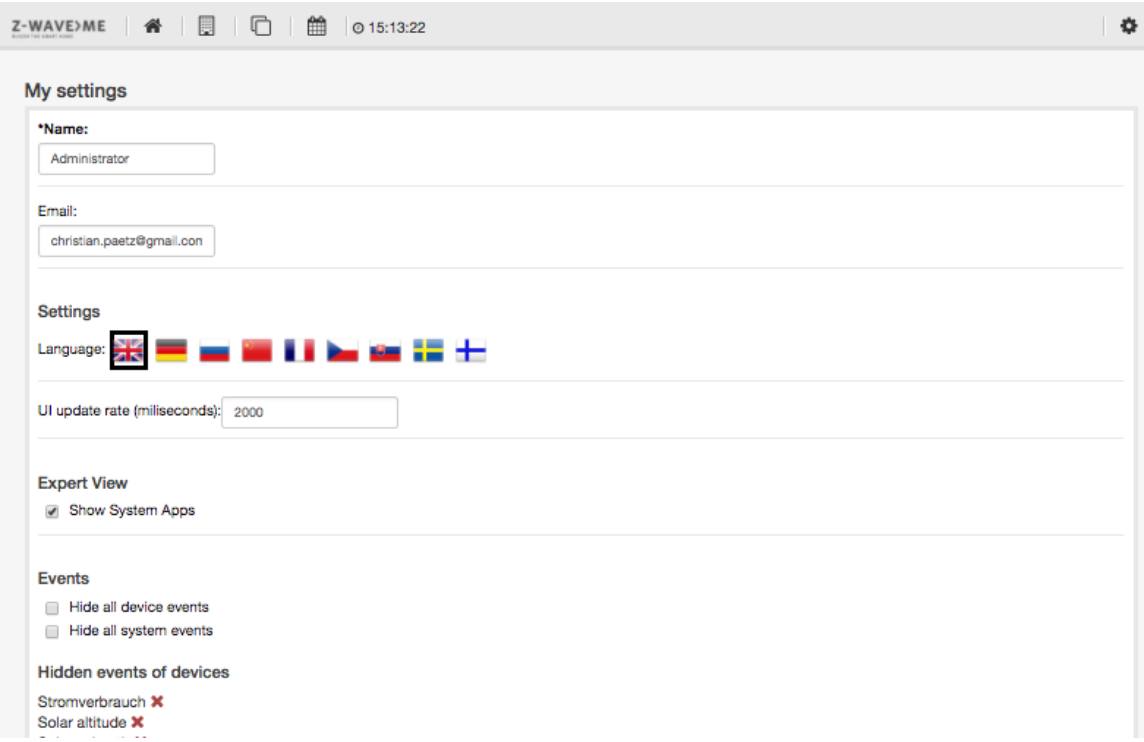


Figure 4.33: My Settings Dialog - upper part

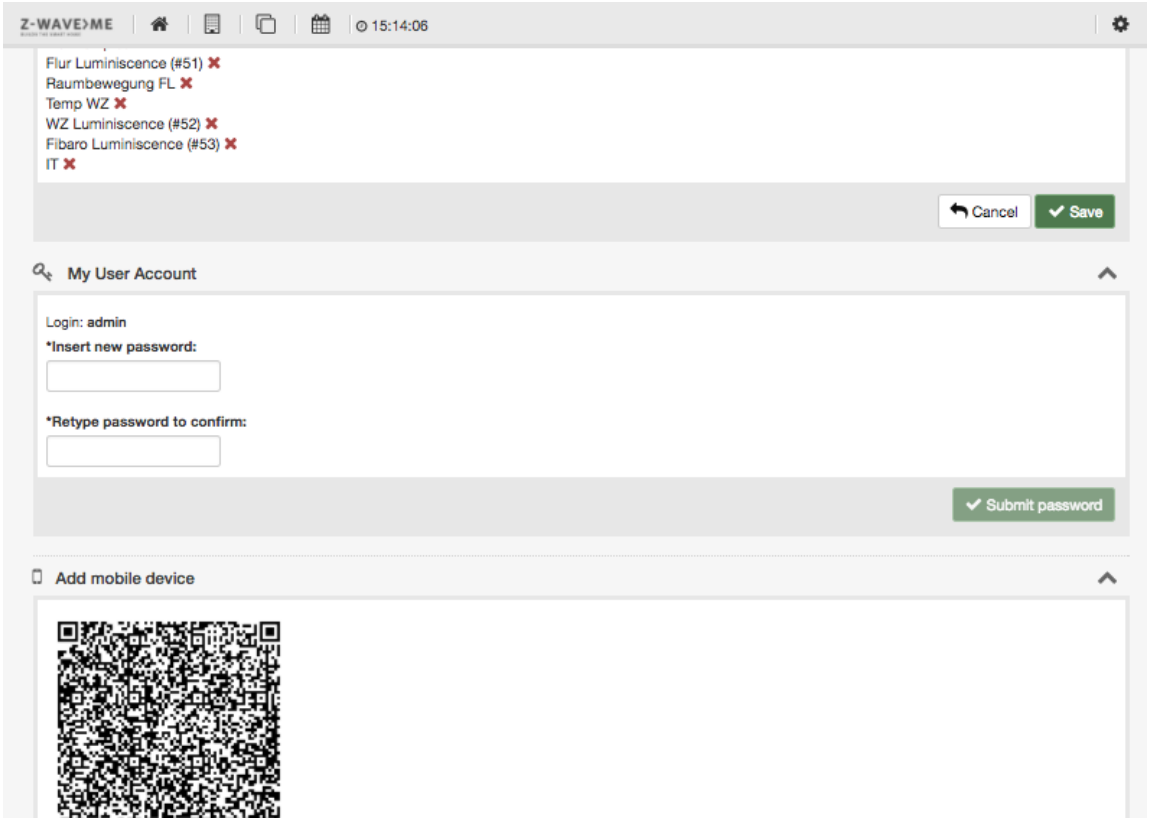


Figure 4.34: My Settings Dialog - lower part

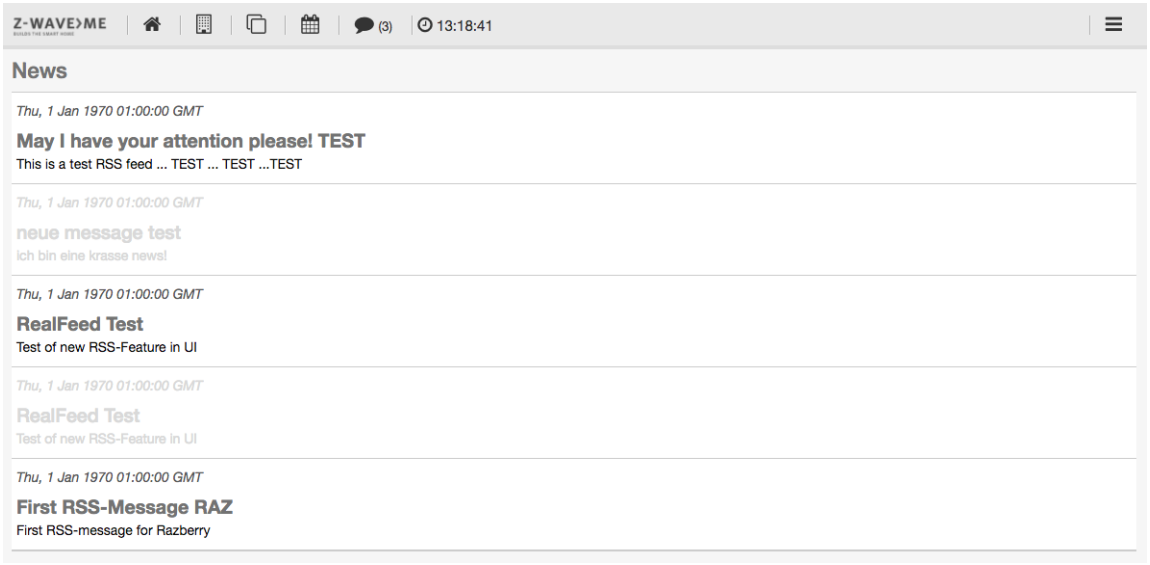


Figure 4.35: List of Z-Way news

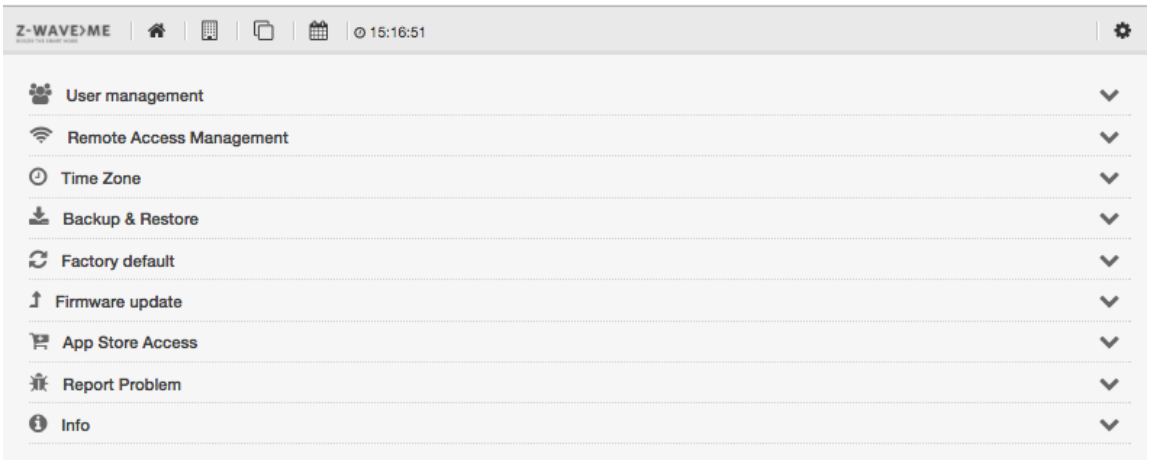


Figure 4.36: Administrator Management Menu

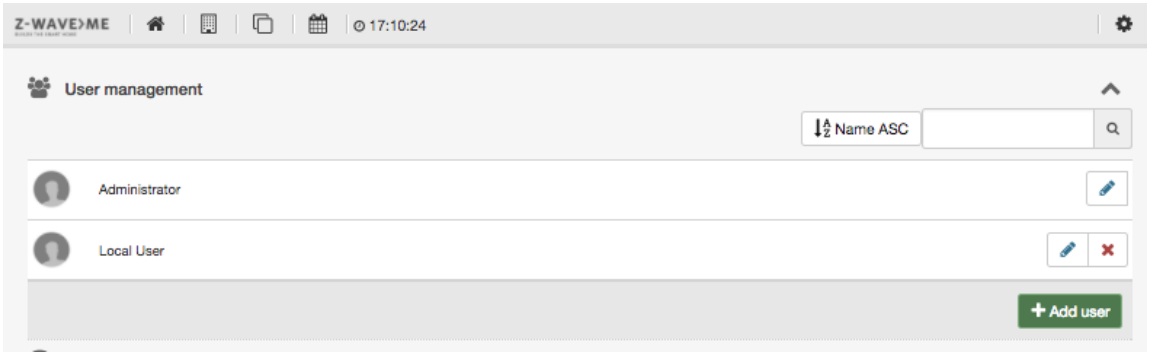


Figure 4.37: User Management

4.3 The Management Interface

Figure 4.36 shows the controller management menu. **Please note that this menu is available for users with administrator privileges** only.

4.3.1 User Management

Figure 4.37 shows the list of users. It is possible to add more users, change their settings, and remove them. Clicking on the setup button opens the user setup dialog. It allows managing

- Name and Email
- Role types: This defines the access rights of the user account. Admin allows accessing the management sub-menu. Standard users will be role-type "users." It is possible to further restrict a user account to real local access or to make it an anonymous user.
- Language of the user interface
- Password settings

4.3.2 Remote Access Management

The dialog shown in Figure 4.38 allows managing the remote access functions of the controller. By default, the remote access function is activated. This enables accessing the controller from any end device in the Internet, e.g. a mobile phone. Please refer to Chapter 3.3 for information on how this remote access is implemented and what security implications this function has.

The remote support function is deactivated by default. This function allows support staff with access rights to the remote access server to remotely access your controller using remote shell (ssh). For complicated support issues, the support staff may ask to activate this function. Please make sure to deactivate it after the session. The remote ssh is only accessible to support staff with support infrastructure rights. Nevertheless, there is no good reason to keep a port open if not needed.

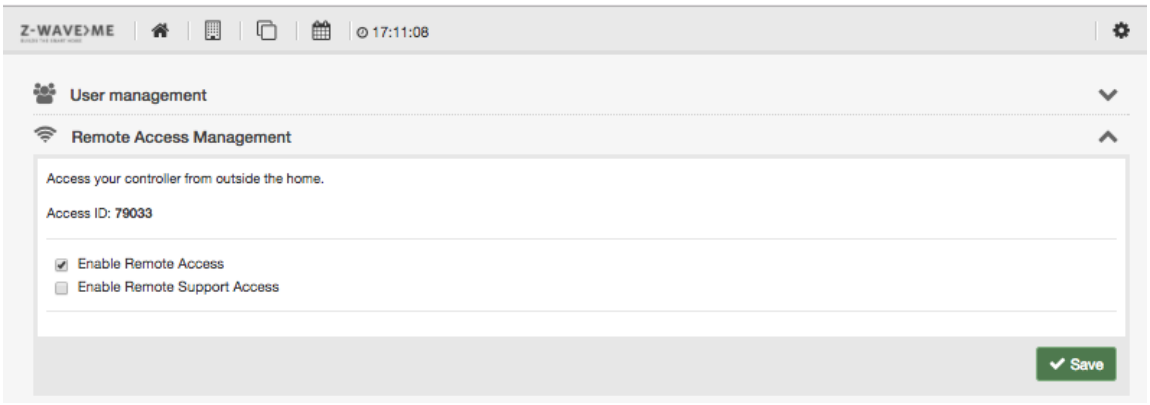


Figure 4.38: Remote Access Management

Figure 4.39: Time Zone Management

4.3.3 Time Zone

The dialog shown in Figure 4.39 allows managing the time zone of the controller. This ensures that all time stamps and the time clock on the top menu bar refer correctly to the local time at the location of the server. Please note that the time remains unchanged if you access the device from a browser from a different time zone.

4.3.4 Backup & Restore

Backup and restore can be done in two different ways:

- Time driven and automated into the Z-Way cloud service.
- Manually triggered into filesystem of PC running the web browser.

The first block of the Backup & Restore dialog controls the cloud backup, which can be activated or deactivated, as shown in Figure 4.40. When activated, the controller will automatically generate a backup file and send it to the cloud server using SSL encryption.

The files are stored on a server managed by Z-Wave.Me. This is a convenient way to keep and update a backup file—and its free of charge. However, if you don't trust this server or the company, just don't use cloud backup!

The cloud backup interface allows defining the backup interval and the notification in terms of failed or successful backup performed.

The local backup option, as shown in Figure 4.41, will generate a local backup file that needs to be stored on the local hard drive of the PC running the web browser. The file is identical to the file stored in the cloud. From the technical side, this is a ZIP file that can even be decompressed and audited. It will comprise XML and JSON files and images that were uploaded before.

If the cloud backup option was chosen, the backup file needs to be downloaded to the local PC before being applied as a restore file. Clicking on the button "Request Cloud Backup" will cause the server to generate a temporarily valid token which is sent by mail to the mail address defined for the admin account. This email will contain an explanation of the process and a unique link to an online list of backup files available. Just download the file of choice.

The real restore function always requires a file uploaded from the local file system. A checkbox ensures that the user understands the consequences of applying a restore file.

Please note that the backup and restore function will only handle files on the controller, and not the Z-Wave network

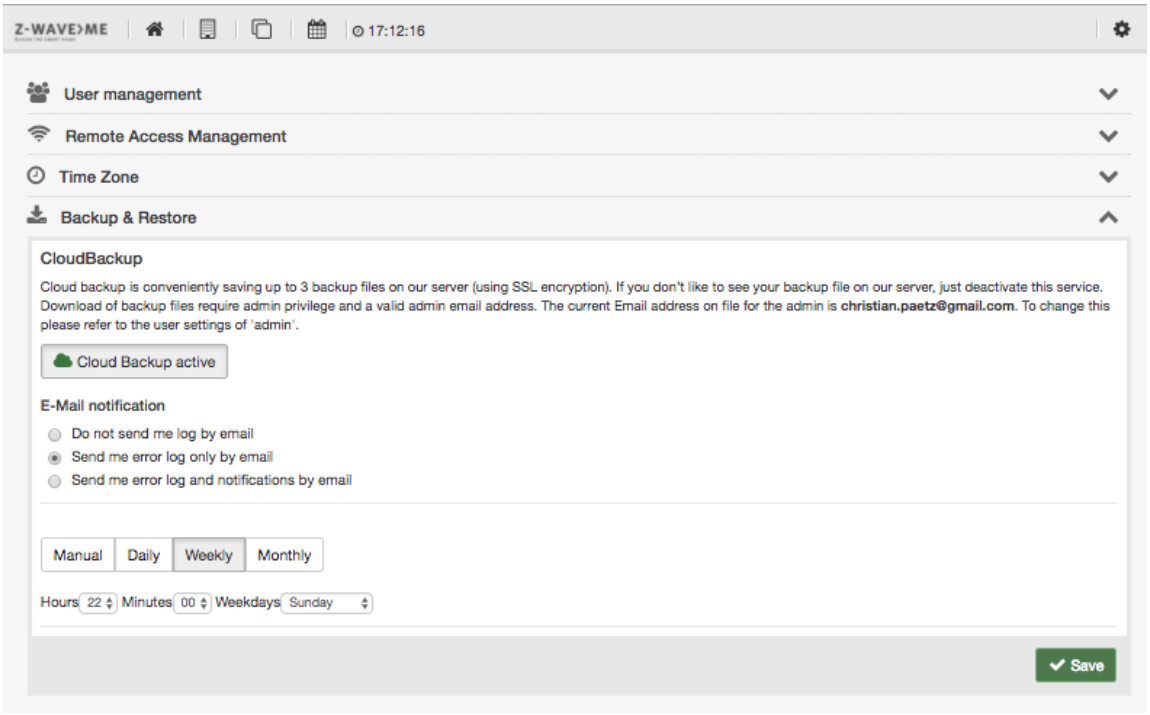


Figure 4.40: Automated Backup into Cloud

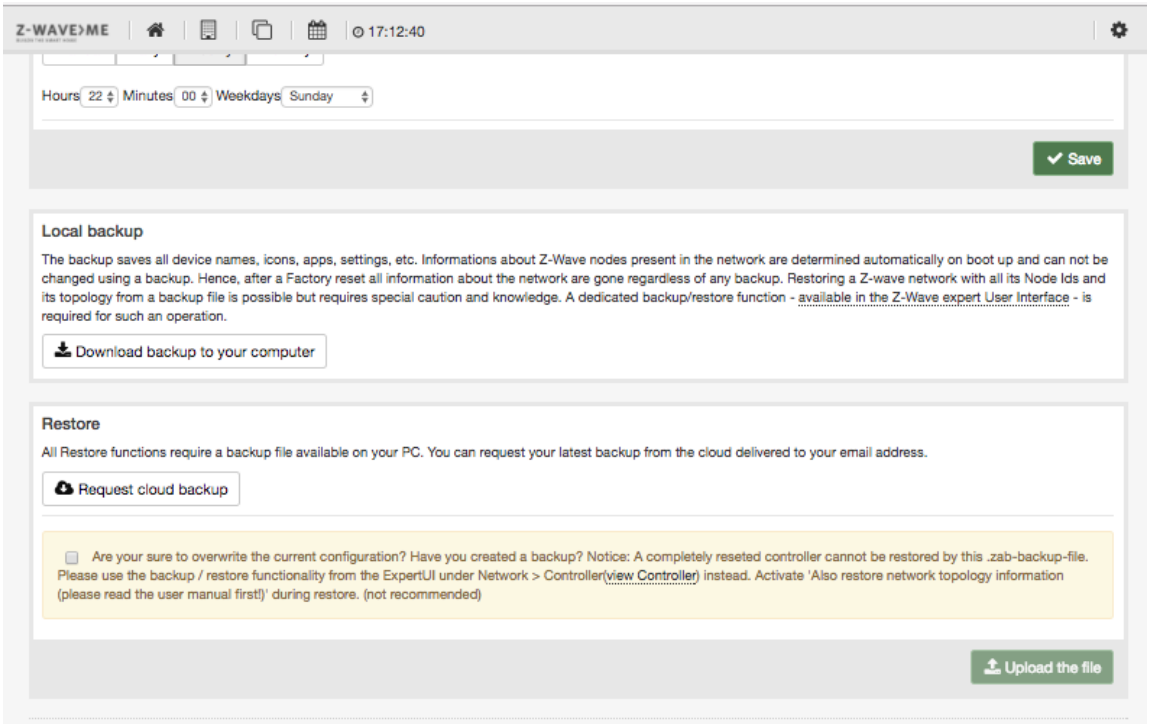


Figure 4.41: Local Back and Restore

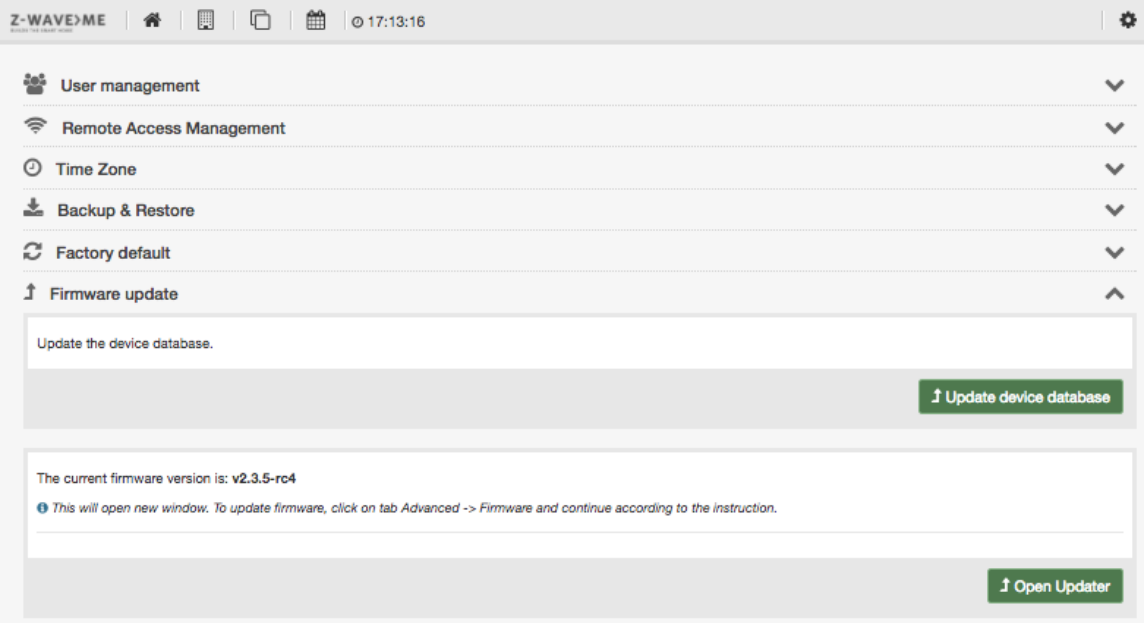


Figure 4.42: Firmware Update Options

topology stored in the Z-Wave transceiver chip.
To overwrite this content, please refer to the **Z-Wave Expert User Interface** , as described in Chapter 7.5.1.

4.3.5 Factory default

This function resets all functions of the controller. All uploaded images will be deleted and all given names and settings will be removed. The included Z-Wave devices will NOT be removed. For removing them from the controller, please refer to the basic Z-Wave literature, e.g. the book “Z-Wave Essentials” as mentioned in Chapter 1.

4.3.6 Firmware Update

The firmware update menu as shown in figure 4.42 refers to two different processes:

- Update device database : This button can be triggered to update the Z-Wave device database used for the inclusion process described in Chapter 4.2.2. This update is not critical and new firmware updates will update this device database anyway. However, for debugging purposes, it is sometimes beneficial to force a database update.
- Firmware Update : This option will update the whole firmware including the dialog offering this update.

Updating the firmware is a quite complex process. In the normal operation mode, all communication between user interface and controller backend is handled using IP Port 8083. However, there are good reasons not to use this port for the management of the firmware update:

- The update script will overwrite the same software that informs about the update as such.
- In case the update fails, or the update firmware is damaged and not working correctly, there is no way to turn the update back since the dialog doing so (on port 8083) will not be available anymore.

This is why hitting the button Open Updater will open a new user interface embedded into the current interface. Figure 4.43 shows this new black-background user interface dedicated to the firmware update function. This user interface is served from another webserver temporarily active on IP port 8084. Hence, it would be possible to directly access this user interface using the

```
http://MYIP:8084
```

.
This user interface will remain active for about 10 minutes. This is enough time to perform the firmware update and revert it in case of problems. After the 10 minutes, the service on 8084 is deactivated automatically.
The firmware update dialog shows the current firmware and offers update to the most recently released firmware version. For debugging or trouble shooting purposes, it is possible to load a specific firmware version. A change log shows the changes of all the official firmware release versions.

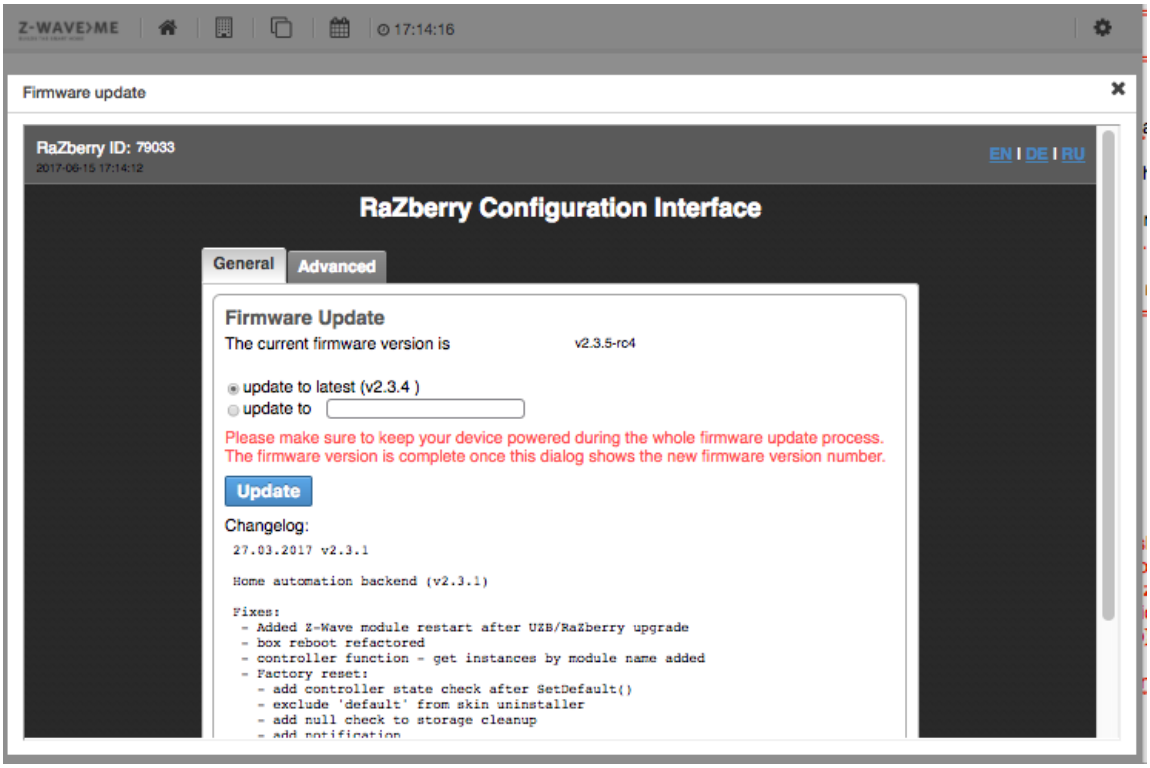


Figure 4.43: Firmware Update Dialog

4.3.7 App Store Access

The app store allows downloading new apps provided either by Z-Wave.Me or other independent developers. Some of these developers may want to limit the use of their apps to a certain group of people, either because this is related to their business model, or because the apps are beta stage or for trial only.
The token concept is a simple and efficient method to limit access. Apps that are uploaded to the app server (For more information on how to create and upload apps, please refer to Chapter 6) can be marked with one or more tags. These tags are simple strings and the developer can choose whatever token he wants. He can also have more than one tag for different purposes.
In order to access apps that are tagged, the various tags need to be added to the controller using the form shown in Figure 4.44. Tags can be added and removed. Once a tag is added, all the apps with this tag will be shown in the app store, as described in Section 4.2.1.

4.3.8 Report Problem

The menu item, as shown in Figure 4.45, demonstrates a quick and simple way to report bugs and problems related to this User Interface. Providing an email is optional, but please be aware that the form will transmit some meta data such as version number of the User Interfaces or version number of the firmware. If you don't like to share such information, please use your personal email. Also, please don't expect an individual answer to the bug reports. This is not a support tool.

4.3.9 Info

The info menu item provides some version information for the user interface and the backend. This information is usually needed for support and troubleshooting purposes only.

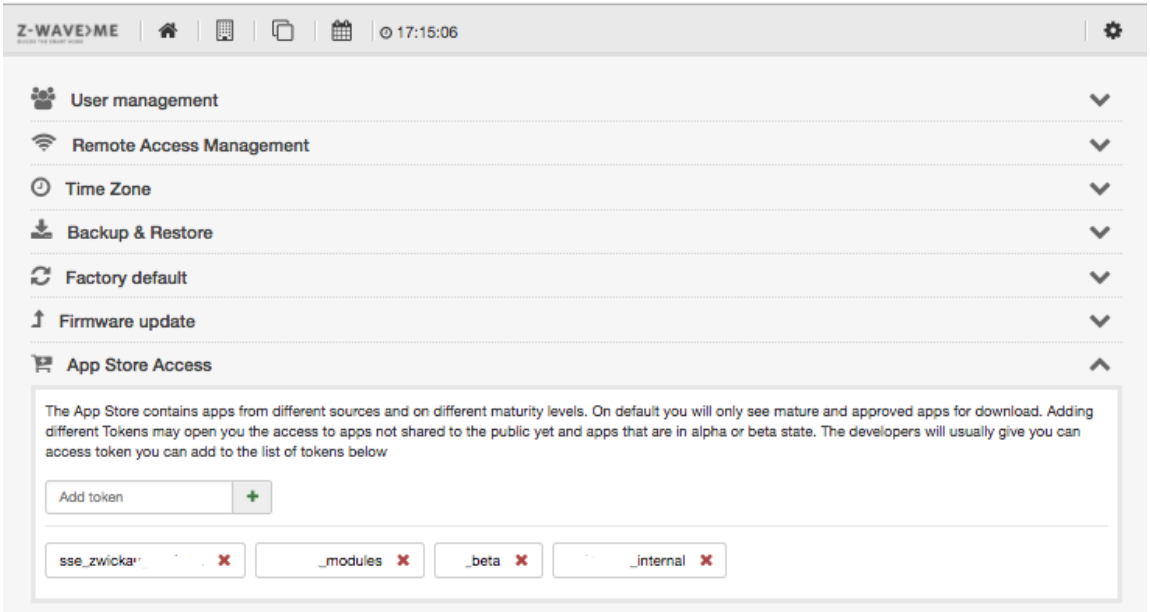


Figure 4.44: App Store Access

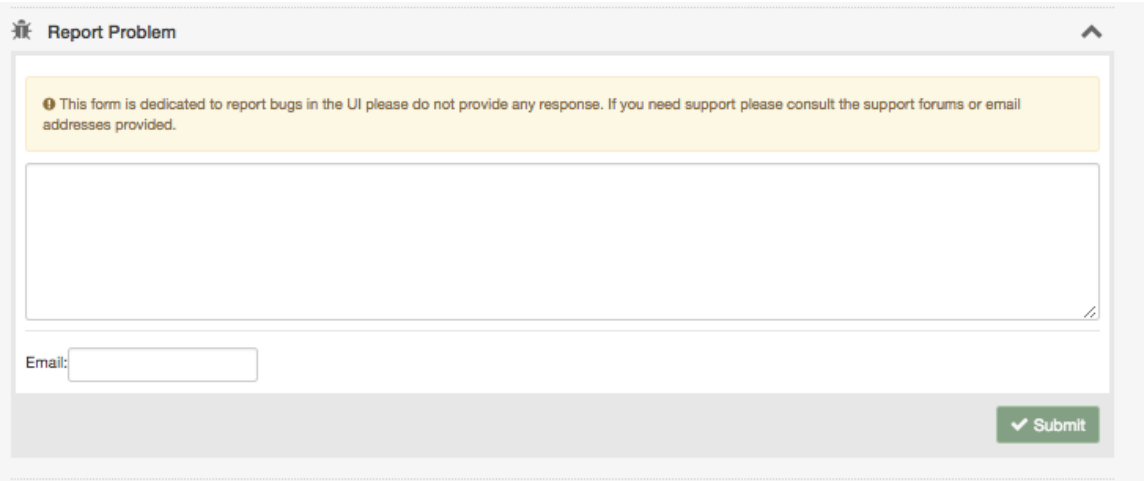


Figure 4.45: Problem Reporting Form

5 Mobile Apps

Accessing the Smart Home from a mobile device such as a phone or an iPad has become popular. Z-Way offers multiple ways to access a user interface from one of these mobile devices. Thanks to an open API, it is also possible to design your mobile app or use third-party apps supporting Z-Way.
Hence, while other vendors offer just one app per mobile platform (Android, iOS), Z-Way allows you to choose what you like best.

5.1 Standard mobile web browsers

The Z-Way Smart Home Interface and the **Z-Wave Expert User Interface** are developed as a responsive design, meaning that the web pages will determine the screen size of your device and re-render accordingly. As a result, the Z-Way Smart Home Interface and the **Z-Wave Expert User Interface** described in Chapter 7 are quite usable with small mobile screens. Figure 5.1 shows a dialog from the web browser interface on a small mobile screen.
The Z-Way Smart Home Interface will always be the most advanced and most recent interface incorporating all new functions and features. This means accessing the user interface on a mobile device with a standard browser makes these functions available on the mobile device first.
The big disadvantage is that the interface is certainly slower than a native app, and the dialogs and menu items may only be partly optimized for small screens.
Accessing the user interface from a mobile web browser is not much different than from a PC-based browser. Just type in the IP address of the controller or use the find.z-wave.me service.

5.2 Native HTML based apps

It is interesting that most apps available in the commonly used app stores are written in HTML and embedded into a native wrapper that allows enhancing the HTML pages with further functions. The standard Z-Wave.Me apps in the iTunes and Google Play Store follow this approach. You find the apps here:

- Android: https://play.google.com/store/apps/details?id=me.zwave.zway
- iOS: https://apps.apple.com/us/app/z-wave-me/id1513858668

The apps are optimized for mobile devices and apps, e.g. to use the QR code for a fast setup. They will also cache data better and align some other functions. There is only one drawback. If the app is used for inhouse control, such fast access to lights, the overall design of the data handling and caching will still imply some delays. Also, the app only allows managing one home per device since it cannot be installed twice for two different homes.

5.3 Pure Native Apps

For the Android platform only, there is a second Z-Way control app that is written completely in Java for very fast access to actuators. The usage philosophy is similar to the other apps with dashboard, room view, timeline, history, etc. Figure 5.3 gives one view of the app.
The app, named **'Z-Way Control'**, is available on the Google Play Store:

- https://play.google.com/store/apps/details?id=de.pathec.hubapp

5.4 Third-Party Apps

The outstanding reputation of Z-Way as Z-Wave backend caused many third-party suppliers and developers to support Z-Way with their frontends. The following explanations can only cover a subset of solution and user interfaces. However, the three examples show three very typical approaches.

5.4.1 Imperihome

Imperihome is a vendor-agnostic app that is available for various mobile platforms. It supports quite a long list of smart home devices and systems such as Z-Way. The basic version of the app is available free of cost. The extended

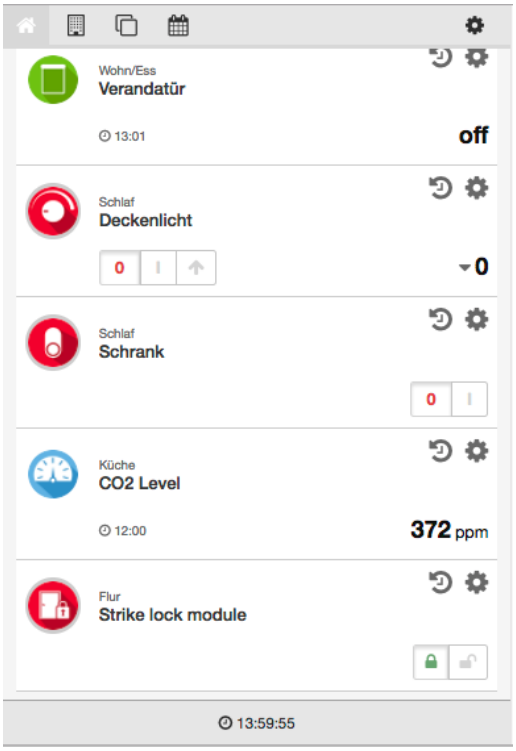


Figure 5.1: Web User Interface on small mobile screen

Figure 5.2: Mobile App Icon from App Store

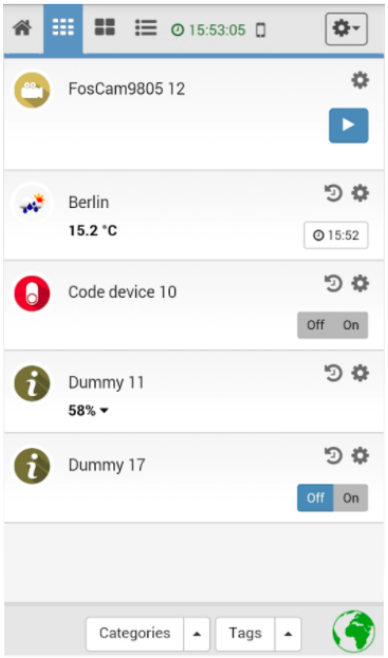


Figure 5.3: Native HTML based app

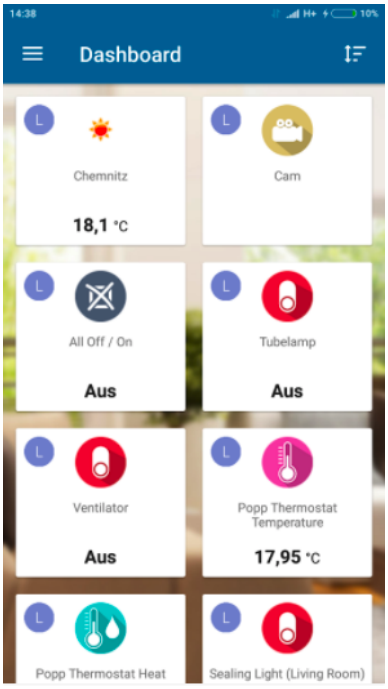


Figure 5.4: Native fast app for Android

Figure 5.5: Imperihome App

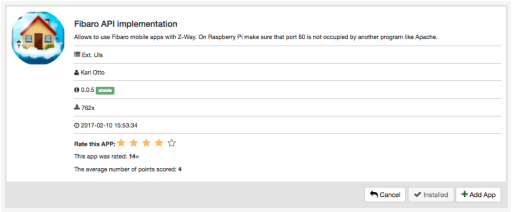


Figure 5.6: Z-Way app to support Fibaro Mobile App

version costs some 5 USD/EUR.

5.4.2 Fibaro

Fibaro is one of the leading Z-Wave-based smart home suppliers in Europe. They offer their own Z-Wave smart home controller, Home Centre Lite of Home Centre 2. Fibaro is focused on design and so their mobile app is quite stylish too.
You can use the Fibaro app, which is designed for their own Home Centre 2, to control Z-Way.
Just download the Fibaro supporter app from the Z-Way app store. Chapter 6 describes how to do this. After that, download the Fibaro mobile app from iTunes or Google Play Store, and configure accordingly.

5.4.3 openHAB

openHAB (www.openhab.org) is one example of an open source smart home control system that uses Z-Way to manage the Z-Wave network part. Once the connection is set up, it is possible to use one of the mobile openHAB apps to control Z-Way-connected devices. For more information about this binding between Z-Way and openHAB, please refer to the openHAB website.

Figure 5.7: Fibaro Mobile App

5.5 Shortcuts for Android and Integration into Third party software

All access to the Z-Way backend can be done using simple HTTP request doable from any webserver. However the client needs to be authenticated. For more information about authentication and client access please refer to chapter 11 and particularly 13.1.

To access the UI with a one HTTP command you need to have a small script on your server. A PHP version of such a script is available at

`http://www.z-wave.me/download/zcmd.php.zip`

.

Please unzip and adapt - if needed - to your needs. We have installed the very same script at

`https://service.z-wave.me/zcmd.php`

Please note that this script - if not running on your own server - bears a security risk since all of your commands run on our server. You will need to trust our server and the safety of the script. Hence we encourage you to use and adapt the script and run in on your own webserver if you have any.

One nice application of the single-line access is placing a shortcut on the Android home screen. This allows to execute important functions in the Smart Home with one click without even opening the app.

First, install the Android App 'HTTP Request Shortcuts' by 'Waboodoo'

`https://play.google.com/store/apps/details?id=ch.rmy.android.http_shortcuts`

The you need two important information from Z-Way beside your login credentials:

- The name of the device: Click on the Configuration Menu of the element you want to control. Element names look like `ZWayVDev_zway_83-0-37`
  where 83 is the Z-Wave device Id, 0 is the instance, 37 is the command class Id (here Switch Binary).
- The command you want to execute. For switches or dimmers it is 'on' and 'off', for a door lock its 'open' and 'close'.

Now start the app 'HTTP Shortcuts' and add a new shortcut. Choose name, icon and description. Pick 'POST' as Access method and enter the url

`https://service.z-wave.me/zcmd.phporthescriptonyourserver`

.

For security reasons we strongly recommend to use HTTPS only. HTTP is not protecting your credentials. Leave authentication and request header untouched. Using the button 'Add parameter' add your credentials and the com-

mand:

- id: boxId/login:password
- cmd: deviceId:command

Then save the settings and try to execute it by hitting the icon from the list of Shortcuts. Once everything is ok, make a long push to the icon and choose 'Place on home screen'. On some phones you need to long push on an empty place on the home screen, then select widgets, find 'HTTP Request Shortcuts' among the list and then select the desired shortcut to show.

6 The App System: making it intelligent

Z-Way operates on two levels: Every function of the device in the network will be shown as an element. Elements are shown automatically but can be deactivated or hidden. All other functions are realized and managed in apps. These apps can be grouped into four categories.

1. More elements that use services and information from the Internet or other TCP/IP-connected devices from third parties. Examples of this include weather information taken from online weather services or the control of a SONOS music system. There are also settings for out-of-band communication to users utilizing push notification, email, SMS, and voice output.
2. Logical connections between elements and other services. This is usually referred to as automation. It connects element functions with each other or with timer information. Example are a time-driven control of lights, heating, or turning on or off the light based on a motion detector.
3. Connection and integration of third-party smart home systems and technologies. One of the most commonly known systems is Apple Homekit. Other systems are openremote.org, IFTTT, etc.
4. A big number of apps is needed to unlock special functions of physical devices (or fix-specific bugs). Examples are the control of user codes and user accesses of a keypad or special displays of energy consumption that are not done by Z-Way by default.

Some important apps are already pre-installed on Z-Way. Most of the apps are available online on the server and need to be downloaded before they can be used. This chapter gives some examples and recommendations for typical apps available.

6.1 A simple Apps as starter - 'Local Weather'

Displaying the local weather inside the smart home user interface is a simple and popular task. Z-Way offers several apps for this request. Already on the device you fine the app **'Local Weather'** calling data from the known service openweather.org. The data is provided free of charge; however, an API key is needed to access the data. Visit www.openweather.org and register as a user to access your API key. After that, install the weather app **'Local Weather'** (see Figure 6.1) from the local app repository. Figure 6.2 shows the configuration dialog:

1. Rename the app to your own needs
2. Pick the name of the city
3. Pick the country of this city (agreed this is foolish, but this is how openweather accepts data)
4. Choose between Celsius and Fahrenheit
5. Insert the API key received from openweather.org

Once activated, a new element is shown on the element view displaying the temperature and the weather situation. Clicking on the small triangle on the right-hand side opens a small window with more weather information such as air pressure, wind speed, relative humidity. The data is updated hourly.

It is possible to display certain values as single element in the element view. This can then be used for automation. Please note, that both the IF and the THEN side of automation like **'IF->THEN'** must always refer to active elements. for more information about the **'IF->THEN'** app please refer to Chapter 6.2.2.

An interesting app is **'Virtual Rain Sensor'**. This app creates an app indicating if it was raining on the location. This app sits on top of a weather app and uses their functions and setup. This example shows that certain apps can depend on other apps to be installed first. This concept is known from PC software where certain applications require certain tools or libraries installed first.

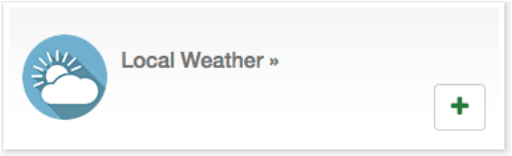


Figure 6.1: The Open Weather app in the App Repository

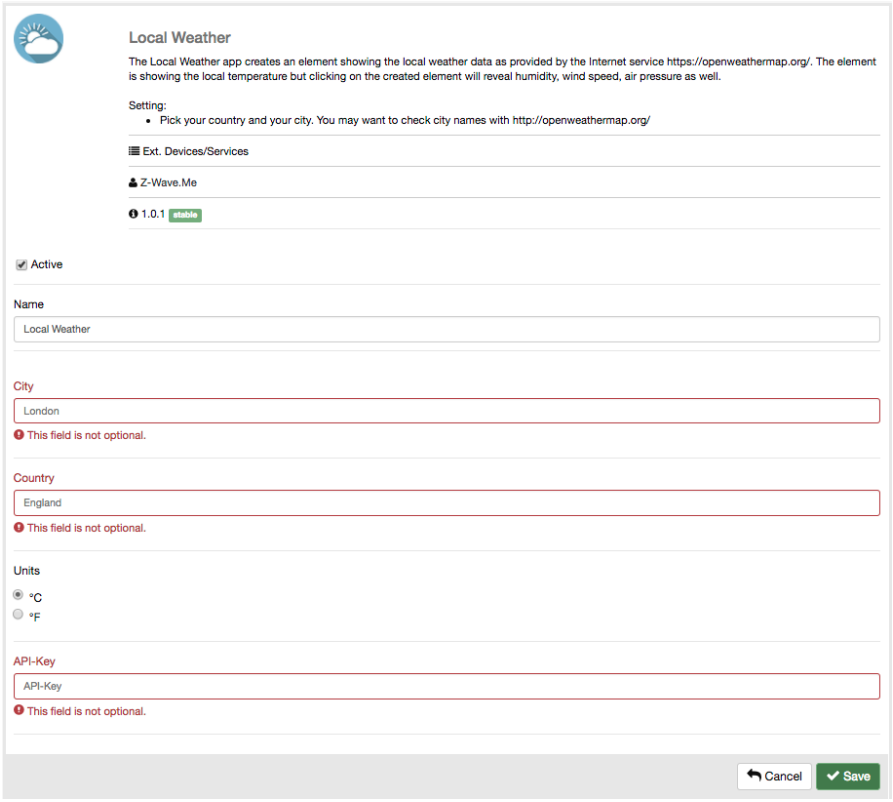


Figure 6.2: The Open Weather app configuration

Figure 6.3: The Scene App

6.2 Smart Home Logic

Logic or automation is the core of the Smart Home. It allows doing things automatically depending on certain conditions.

6.2.1 Scene

The most basic step to simplify life is to group multiple actions into one action. There is no need to have a smart home controller in the home to do this. Even classical electrical wiring allows switching on two lights with one switch. However, smart home allows creating much larger actions just triggered with one (virtual or real) button. The tool for this is called **'Scene'** and the app is shown in Figure 6.3.

The configuration of the scene app is quite straightforward. All devices to be controlled with the "one button" need to be identified and their desired switching status defined.

The scene itself becomes a virtual device, which is why it is also possible to create a hierarchy of scenes and to let one scene switch the other scene.

Once stored, there is a new element with the name of the scene as shown in Figure 6.4. There is only one button to turn on the scene. A scene can never be turned off but only replaced by a different scene. The reason for this is that it is not reasonable to turn back to the previous state, since individual devices of the scene may have been operated in the meantime. The scene element shows the time stamp when this scene was activated the last time. The event history shows all the events (activations) the scene, much like how it is done on any other Z-Way actuator.

An enhancement for the scene app is the **'Scheduled Scene'** or **'Schedule'**, as shown in Figure 6.5. This app combines the scene function with a simple weekly calendar that allows executing the scene at a special time per day.

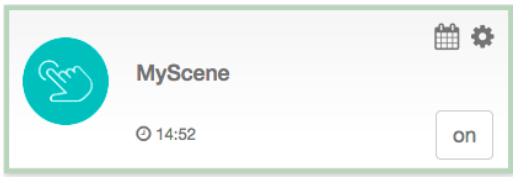


Figure 6.4: The Scene Element

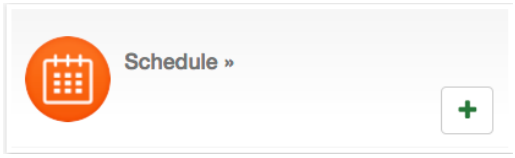


Figure 6.5: Schedule - an scheduled Scene

6.2.2 If -> Then

The most basic relationship of automation is the If->Then relationship

Some examples

> **IF** button 2 on the remote control is pressed, **THEN** the ceiling lamp will turn on. **IF** temperature sensor goes above $22°C$, $\rightarrow$ **THEN** turn down the heating **AND** open the window.

In order to accomplish this kind of IF$\rightarrow$ THEN relationship the following requirements need to be met:

- The actor device needs to be identified and able to perform the desired task.
- The sensor or controller needs to be able to generate an event that causes the action.
- The sensor or controller needs to know which actor to control and in which way in case of an event.

The first requirement is quite obvious. If the ceiling light—to stay in the first example—is turned on, the ceiling light needs to be controlled by a wireless device that can be turned on and off wirelessly. While this sounds straightforward, there are plenty of examples where the actor is not able to fulfill the desired task, e.g. a dimming device cannot change the color of an LED light.

The second requirement is also obvious. There must be a defined event that causes an action. In case a button of a controller is involved, this is quite easy, but for sensors that measure constant values, this may become a challenge.

Binary sensors such as door sensors or motion detectors generate an event whenever their binary state changed from on (window open) to off (window closed). For a motion detector, it gets more complicated. The motion part, typically resulting in an ON event is easy to detect but how about the OFF event?

How can a motion detector be sure that there is no person in the room anymore? Most motion detectors allow setting a certain timeout value and generate an OFF event when the time has run out. It is also conceivable to do nothing after a given time. Even then, the motion detector needs to know the minimum time between two events to be generated. Otherwise, it will constantly generate events, resulting in network traffic when a person moves in the room.

Timings and settings are typical configuration values of a motion detector and often can be changed either locally using buttons and/or wirelessly using the `Configuration` command class described within the **Z-Wave Expert User Interface** in Chapter 7.

Sensors that measure an analog value such as temperature, CO2 level, humidity, etc. cannot generate an event from just measuring the value. In case the device is used to start an IF (...) $\rightarrow$ THEN (...) association action, it needs to know certain boundaries of the measured values and what to do if the measured value reaches the boundary value set. The boundary values that are used to generate events are called **Trigger Levels**.

The **'If->Then'** app, as shown in Figure 6.6, allows implementing the third condition, the relationship between event

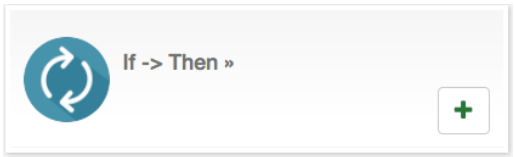


Figure 6.6: If->Then App

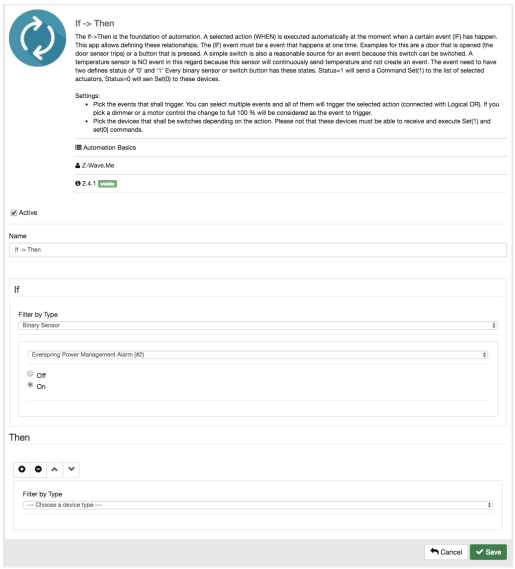


Figure 6.7: If->Then App Configuration Dialog

and action.

Figure 6.7 shows the configuration interface of the **'If->Then'** app. The first step is to device the event. First, select the device type of the relevant device. This will only shorten the list of devices (elements) to choose next. The device types are

1. Binary Sensor: These are typically motion detectors, smoke detectors, door sensors, but also alarm conditions of the device, e.g. power loss.
2. Binary Switch: These are all switches just knowing the state on and off.
3. Multilevel (Analog) Sensors: These are sensors measuring a certain physical value, e.g. temperature, CO2 level.
4. Multilevel Switches: These are dimmers and motor controls, e.g. for blinds or jalousies.
5. Scene Controller: These are special devices like remote control issuing special scene activation commands. The specific scene control number must be known.
6. Switch Control (On/Off/Level): These are controlling devices that report a status of the buttons, e.g. on or off.
7. Switch Control / Scene: These are controlling devices that only know one state. Typically, these are buttons that are pressed. The "un-press" is not monitored.

Finally, choose the event that will trigger the **'If->Then'** rule. For devices with a defined number of states (binary sensor, discrete sensor, binary switch, etc.), reaching this state is the trigger condition. Here it is enough to just pick the state (e.g. off). For analog sensors, it must be defined if the event is reached when the measured value is above, equal, or below a certain trigger value.

Please note that some battery-operated sensors update their value only infrequently. The temperature on a certain spot may rise. Unless the sensor does not transmit this new value, the **'If->Then'** rule will not kick in.

The second part of the configuration is defining the action. The device and action selection is similar to the IF part. Choose the device type first, then the specific device (element), and finally the action. The selection of device types differs from the IF section for obvious reasons:

1. Binary Switch: These are all switches just knowing the state on and off.
2. Color Switch: This allows changing the color on multicolor lights.
3. Door Locks:
4. Multilevel Switches: These are dimmers and motor controls, e.g. for blinds or jalousies.
5. Scene: A scene as described in Section 6.2.1
6. Thermostat: This allows defining the setpoint of a thermostat.

Once saved, the configuration becomes active.

One special function of computers in general, and the **'If->Then'** app in particular, is that it does not necessarily know what the user thinks but what he configures.

Let us take an example: If the relationship is that—**IF** the door sensor is open, **THEN** turn on the ceiling lamp—this means that the ceiling lamp goes on when the door is opened. When the door is closed, the ceiling lamp will not go off because this was not written. If the ceiling lamp goes off, a second instance of the If->Then relationship is needed. In case two devices really run synchronously with triggers on and triggers off, another app can realize this step instead of having two times the If->Then app. This app is called is called **'Association.'**

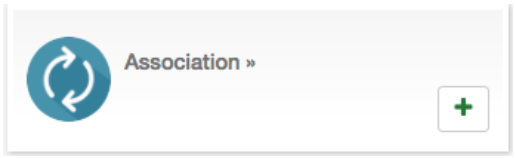


Figure 6.8: Association App

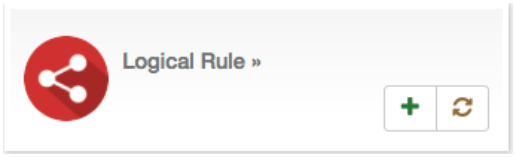


Figure 6.9: Logical Rule

6.2.3 Logical Rule: If->Then on steroids

'If->Then' can connect one event with multiple actions including triggering a scene with even more action. However, it is always one single event. For example, the night light will be turned on by a motion detector but only in the evening and not during the day. This means that different input variables need to be combined to generate the final scene-triggering event. The way this combination is achieved is called binary logic or Boolean logic, and the app implementing this is called **'Logical Rule'** as shown in Figure 6.9. Boolean logic has three basic ways to combine variables:

- **AND**
- **OR**
- **NOT**

With these three elements, even complex relationships between variables can be described. In case of the evening light triggered by a motion detector, the definition looks quite simple:

IF (it is evening) **AND** (Motion detector triggers) → **THEN** (activate scene)

It is possible to connect more than two input variables using Boolean logic. However, some constrains need to be considered.

- The logical combinations, namely AND and OR always combine two variables. If more than two variables are combined, there is a need to set braces: The statement "A and B or C" has two meanings: (1) always A and then either B or C, (2) Either a combination of A and B, or just C.
- There is a difference between status value and events. A scene can only be activated by one single event, but this event can be combined with a list of status value. The scene is triggered only if the event happens and all the other status variables are in the desired status. In case a scene depends on two events, then the trigger condition is only true if both events happen at the same moment, which is quite unlikely.
  A combination of variables therefore always has one single event but is not a limited list of other status values. Status values are "after 17.00" (not right at 17.00, this is an event), a certain switching state of a switch (not the change of the switching status, this is an event).

Figure 6.10 shows the configuration of the **'Logical Rule.'** The first section allows defining certain conditions (status or events) and defines if all of them (AND) or just one of them (OR) will trigger the rule.

A condition can also be the result of another combination of statuses and events. This is called "nested condition," which allows building a hierarchy of conditions and combining them in any possible way.

The action section is already known from **'If->Then'** of from scenes. The third section, "How the Logical Rule is triggered," allows some runtime optimization. By default, any changes in the devices mentioned in the rule will have the chance to trigger the entire rule. For very large rules, this may consume a lot of power. That's why it is possible to limit the number of devices that can trigger the rule. This saves computing power.

6.2.4 Tips and Tricks

Besides the apps **'Scene,'** **'If->Then,'** **'Logical rule,'** and **'Schedule,'** there are a number of other apps in the app store for special automation functions. One little utility is worth mentioning—the dummy device shown in Figure 6.11.

The dummy device creates a virtual switch or dimmer that is shown as an element but does not have any physical

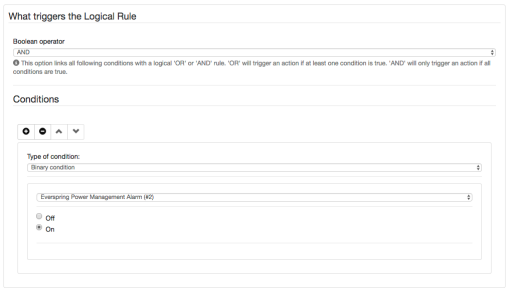


Figure 6.10: Logical Rule

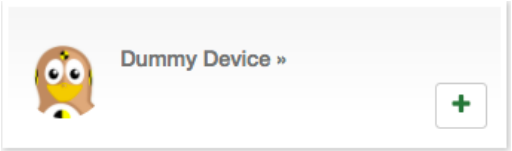


Figure 6.11: Dummy Device

function. Nevertheless, it is a valid source of events and status information as well as a valid device to be controlled by scenes. Sometimes, this is helpful to visualize certain situations in the home.

6.3 The big apps

While automation apps are more or less a toolbox to implement original ideas of certain automation and dependencies, the app store also offers complex apps for certain typical functions in the smart home that are already finished and need configuration only.

6.3.1 Leakage Protection

The leakage protection collects all information from leakage sensors in the smart home and generates one single element to visualize the status of the home. Additionally, the alarm condition is communicated out-of-band. The app needs to be downloaded from the online server, as shown in Figure 6.12.
The configuration allows picking all flood sensors in the home to trigger an alarm. In case of an alarm, certain actions can be triggered. The most obvious action would be to turn off the water supply using a Water Shut Off valve. Additionally, it is possible to send out a notification. A drop-down list allows picking the desired notifier (email, push, SMS, whatever is installed) and define the message to send.
The app creates an element to control the leakage alarm. The element allows arming and disarming the system. See

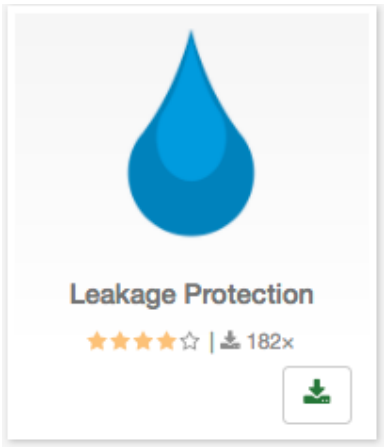


Figure 6.12: Leakage Protection App

Figure 6.13: Leakage Protection element - armed

Figure 6.14: Leakage Protection element- alarm

Figure 6.13 for the element when in the armed status. In case one of the flood detectors detects a leakage, the app will go into the alarm state:

- The element shows the alarm state (Figure 6.14).
- All actions defined in the configuration dialog will be executed.
- If configured, a notification message is sent using the notifier selected.
- The little triangle on the element allows checking which sensor triggered the alarm.

In case the alarm condition disappears (no water anymore), the alarm condition is revoked, but the element will show that there was an alarm event. This indication is shown in Figure 6.15.

6.3.2 Fire Protection

The fire protection collects all information from smoke detectors in the smart home and generates one single element to visualize the status of the home. Additionally, the alarm condition is communicated out-of-band. The app needs to be downloaded from the online server, as shown in Figure 6.16.

The configuration allows picking all smoke detectors in the home to trigger an alarm. In case of an alarm, certain actions can be triggered. The most obvious action would be to turn on all lights and open the door. Additionally, it is possible to send out a notification. A drop-down list allows picking the desired notifier (email, push, SMS, whatever is installed) and defines the message to send.

The app creates an element to control the fire alarm. The element allows arming and disarming the system. See Figure 6.17 for the element when in arm status. In case one of the smoke detectors detects a leakage, the app will go into the alarm state:

- The element shows the alarm state.
- All actions defined in the configuration dialog will be executed.
- If configured, a notification message is sent using the notifier selected.
- The little triangle on the element allows checking which sensor triggered the alarm.

In case the alarm condition disappears (no water anymore), it is revoked, but the element will nevertheless show that there was an alarm event.

Figure 6.15: Leakage Protection element- wait for clear

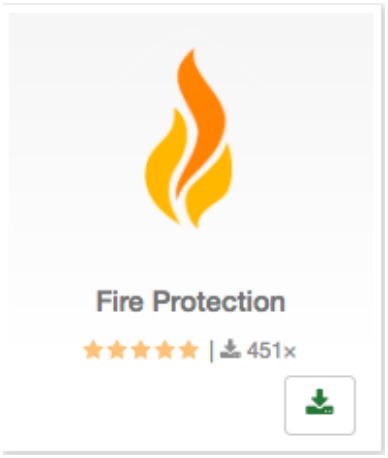


Figure 6.16: Leakage Protection App

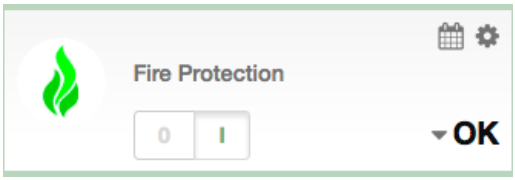


Figure 6.17: Fire Protection element - armed

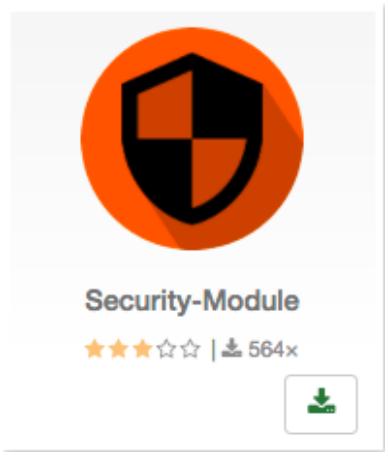


Figure 6.18: Security System

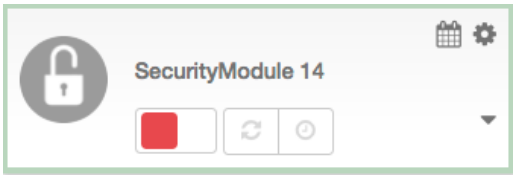


Figure 6.19: Security System im disarm status

Figure 6.20: Security System im arm status

6.3.3 Burglar Alarm System

Security is one of the most frequently used functions of the smart home. The smart home can replace the traditional alarm system and implement the function using dedicated devices or reusing other devices such as e.g. motion detectors that were primarily installed for different reasons.
The app **'Security Module'** implements a complete alarm system with all the functions known from conventional alarm systems. The app must be downloaded from the online server, as shown in Figure 6.18.
The configuration interface allows managing different lists of devices:

1. Devices that can trigger the alarm: These are all the sensors that will indicate a burglar in the home. These include door sensors, motion detectors, tamper switches, glass break sensors, etc. Per device the app allows selecting what sensor state will trigger the action.
2. Devices that can arm/disarm the system and clear alarms: Of course, it is possible to arm/disarm and clear alarms using the user interface. However, most alarm systems are armed/disarmed using buttons, keypads, or even smart home scenes (e.g. "I am leaving home" or "I am sleeping"). For example, a simple switch can be used to arm or disarm the alarm system. This is not safe but doable. Per device an arm, a disarm, and an alarm clear status or event can be defined.
3. List of actions on alarm. This can be turning on lights, starting SONOS, switching a siren, and of course triggering a notification of choice.
4. List of "arm" status indicators: Once the alarm system is armed, there will be some visible indication, besides the element on the user interface, that the house is armed. This could be some red lighting or some slow glowing LED light.
5. List of "disarm" status indicators: Similarly, there can be devices that indicate that the alarm system is disarmed, e.g. with a green light.
6. List of actions when the alarm is cleared:

The last section of the configuration allows defining time-driven arming and disarming of the system.
The security app creates an element to control and manage the alarm system. The element allows arming and disarming. Figure 6.19 shows the alarm system in the disarm state. Once armed, the icon turns blue for some seconds, indicating that the alarm is turned on but the alarm system is not yet fully armed. This is important as it allows users to leave the home after they have armed the system. Any sensor in the list of triggering devices will put the system in alarm state once triggered. This results in

- The element shows the alarm state with the red icon, as shown in Figure 6.21.
- All actions defined in the configuration dialog will be executed.

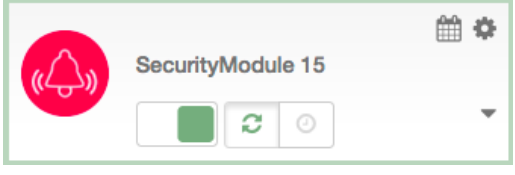


Figure 6.21: Security System in alarm status

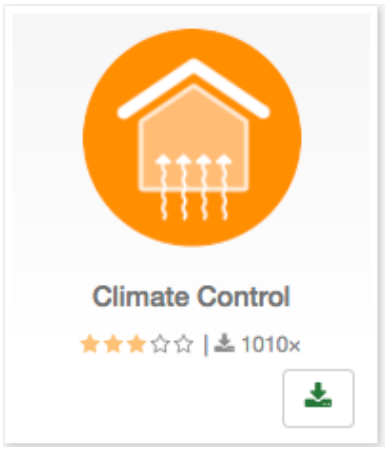


Figure 6.22: Climate Control App

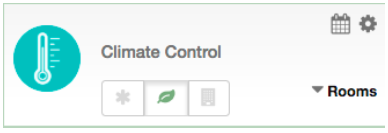


Figure 6.23: Climate Control App Element

- If configured, a notification message is sent using the selected notifier.
- The little triangle on the element allows checking which sensor triggered the alarm.

Even if the triggering sensor goes back into the non-triggering state, the alarm conditions remain active. The alarm must be cleared by one of the devices configured or clicking on the alarm clear button (two arrows). The system then goes back into the arm state, as shown in Figure 6.20. The clock icon will activate the time-driven arming and disarming of the alarm system.

6.3.4 Climate Control

Saving energy by having intelligent heating and climate control is one of the core values of the smart home. Of course, it is possible to directly control thermostats, but the **'Climate Control'** feature manages the whole home and offers a lot of options.

The app must be installed from the app store as shown in Figure 6.22.

The climate app operates on various levels. First, there is a time-driven weekly schedule per room that defines the temperature in that room. The time-driven schedule should be the normal operation mode of the climate control feature. A second layer is the manual overwrite of the temperature. This overwrite can be done on a room layer as well as on the whole home layer.

The first two values are of general nature. The setback temperature is the temperature difference between the comfort temperature in a room and the energy-saving setback temperature in this room. Since different rooms may have different comfort temperatures, the energy-saving temperature also differs but always by the same delta. In case of doubt, please insert 4 Kelvin, which is a commonly accepted value.

The automation reset time defines when the normal automated heating schedule is used again after a manual overwrite of this schedule. The preset 2 hours is a good value.

Now there is a list of rooms. Just add your rooms you want to have the climate control feature. Per room you can define a temperature sensor that shows the temperature in this room in the climate control user interface. There is no further function of this sensor than showing the value in a convenient way. Please note that the dropdown list will only show temperature sensors that are assigned to this room. The comfort temperature is the room-specific temperature. Usually it is higher in bathrooms than in sleeping rooms. Your individual preference matters here.

The last section of the configuration dialog is the heating schedule. It allows setting a temperature at certain time slots on certain days per week for certain rooms.

As shown in Figure 6.23, the app generates a special element. It allows running the climate control for the whole home in three basic modes:

-  The heating in the home is turned off. This will overwrite all schedules and all settings in every room.

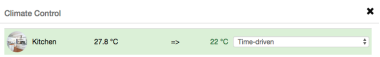


Figure 6.24: Climate Control App Element - room view

- The heating in the whole home is in the energy-saving mode. This will overwrite all schedules and all settings per room.
- No home-wide overwrite. The room-specific settings apply.

The little triangle on the right-hand side allows opening the room view as shown in Figure 6.24. Here, it is possible to see the actual temperature per room plus the current desired temperature. A dropdown list allows choosing the heating mode for the specific room:

- **Frost Protection:** The room is in the frost protection state (around 8 degrees C).
- **Energy Save:** The room is in the energy-saving state. This is the comfort temperature minus the setback temperature difference defined in the configuration dialog.
- **Comfort:** The room is in comfort temperature as defined in the configuration dialog.
- **Time driven:** The heating schedule as defined in the configuration dialog applies.

6.4 Out-of-band notifications

All events in the smart home are shown in the user interface, or they can be indicated using devices inside the home. However, people do not always monitor the user interface. Hence, there are the so-called out-of-band notification options to reach the user in such a case:

- Email
- SMS
- Push notifications right on the home screen of the mobile phone
- voice call

Z-Way supports various ways of out-of-band communication. For every communication channel, multiple apps from different providers may exist to realize the same function. However, all these notifiers work in the same manner:

- They establish an out-of-band communication channel.
- They need to be configured according to the user's preferences.
- They accept messages from other apps and forward them as configured.
- They create an element with a push button to send a simple test message.

Some out-of-band notifiers make it possible to gather and filter events from the timeline and forward them. This must be configured in the configuration dialog.

6.4.1 Push Notifications

Push notifications are delivered to a mobile phone. As soon as one of the native Z-Wave.Me apps is installed on the mobile phone, the push notification option is automatically enabled. Push notifications allow gathering events and forwarding them automatically. This can be configured in the apps **'Mobile Phone Support, Notification Filtering'** which is already running in the system. Please go to Active app management to open the configuration dialog.

6.4.2 Notifications by E-mail

The **'Notifications by E-mail'** app, as shown in Figure 6.25, for each user creates a communication channel for sending events by email.
In the **'Notification Filtering'** application, you can select the notification channel, it can be email, push, sms or a channel provided by third-party applications: facebook, twitter and others.

6.4.3 Other notifiers

Besides the two standard notifiers for push notifications and email, the app store has plenty of other notification apps from third-party developers. Just check out what works for you.

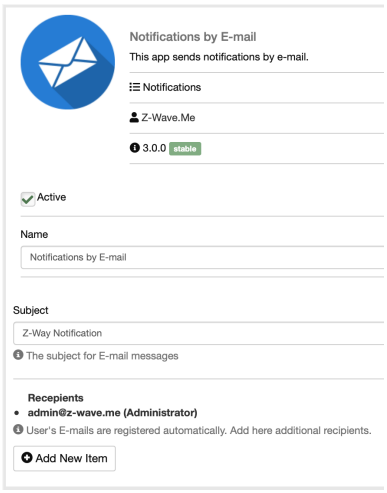


Figure 6.25: Notifications by E-mail

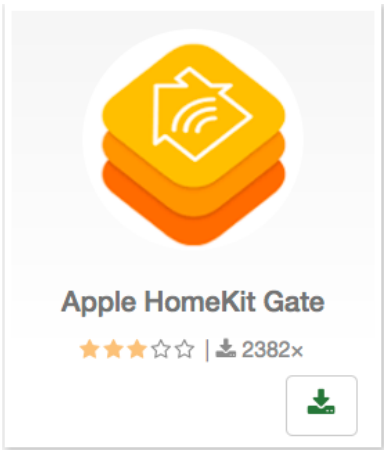


Figure 6.26: Apple Homekit Integration

6.5 Useful tools and utilities

The app store is a gold mine of cool applications. This manual can only mention a few of the more popular ones. Of course, you can check out the display of apps according to popularity, etc.

6.5.1 Apple HomeKit

As shown in Figure 6.26, the Apple Home Kit App, provided by a third-party programmer named Andreas Freud, connects Z-Way with Apple's HomekitWorld. Once installed, the Z-Way controller is shown to Apple as a Homekit bridge device. Please be aware that this app is maintained by a third-party developer and the existence of the apps is certainly not in the main interest of Apple.

6.5.2 Intchart.com

This app (Figure 6.27) adds a little icon to the selected device. Clicking on this icon opens a window with a chart showing its history. To use this app, you have to be registered at www.intchart.com. Note that if you change the settings below, the chart can be reset, but the previous chart will still be visible on intchart.com.
The settings are as follows:

- First, register at www.intchart.com.
- Below that, select the devices to track.
- Indicate if they have to be on the same chart.
- Indicate if you want to have a difference between values (for energy consumption, etc.).
- Choose the poll period.

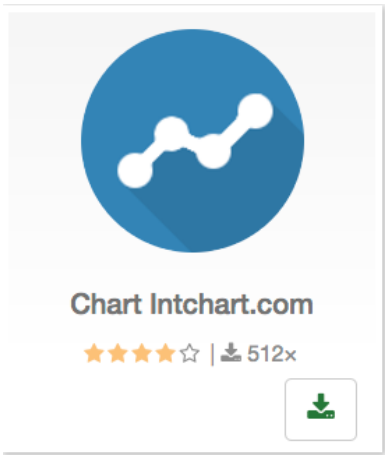


Figure 6.27: Intchart.com Integration

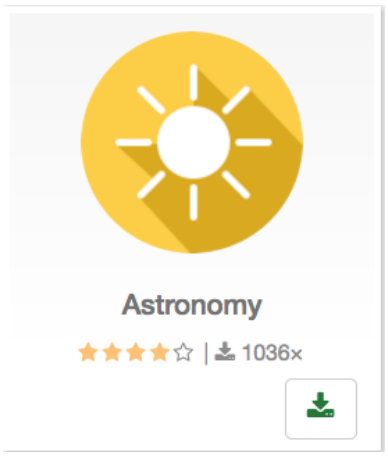


Figure 6.28: Astronomy App

- Paste the API user ID and API key from your account: www.intchart.com.

6.5.3 Astronomy App

This app (Figure 6.28) from Maros Kollar calculates the position of the sun above the horizon for the given location. The module provides various metrics for other automation modules like sun altitude and azimuth, and emits events when the sun reaches certain positions. This module can be used to control light scenes or shading based on the solar position.
Check github.com/maros/Zway-Astronomy for detailed documentation.

6.5.4 Alexa Integration

This app in Figure 6.29 integrates Z-Way with the Amazon Alexa Voice Control system. Once installed, you need to activate the "skill" in the Alexa user interface before using.

6.5.5 Philips Hue Integration

HUE is a new way to use your Philips Hue SmartHome solutions. Use this app (Figure 6.30) as a remote to switch colors, turn up the brightness, and quickly toggle between lights on and off. For the moment, you have to create your credential manually.
Installation instructions:

- Go to http://YourBridgeIpAddress/debug/clip.html
- Enter {"devicetype":"SmartHome#RasberryPi Zway"} in MessageBody part.
- Go and press the button on the bridge.

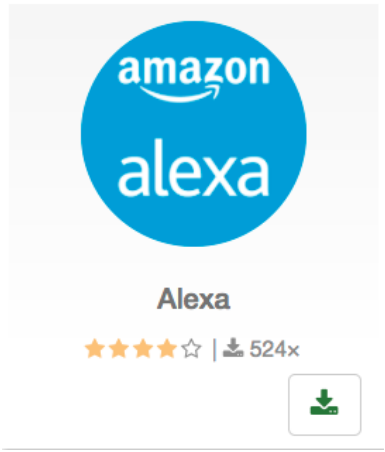


Figure 6.29: Amazon Alex Integration

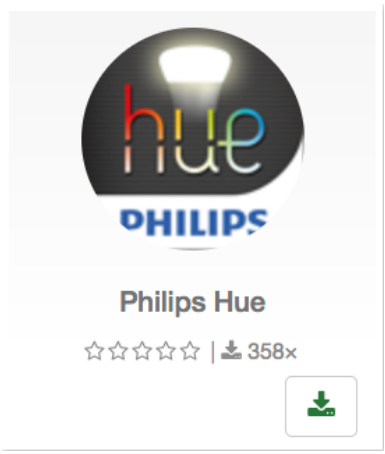


Figure 6.30: Philips Hue Integration

- Press the POST button on clip.html page and you should get a success response.
- Congratulations you have just created an authorized user (like: 1028 d66426293e821ecfd9ef1a0731df), which we'll use from now on.
- Fill your key in the Hue app!

To create your own key, see more details on:

- www.developers.meethue.com/documentation/getting-started
- https://github.com/timauton/Hue

6.6 For Developers

Apps for developers require a certain amount of programming skills and partly require knowledge about the Z-Way data model. Please refer to Chapter 11 for details.
Nevertheless, the app **'HTTP'** device allows adding certain functions without deep software knowledge. The **'HTTP'** device generates a sensor or an actor depending on information obtained by just accessing a website using HTTP.
One example will demonstrate this. The goal is to make an element that shows the current USD/EUR exchange rate.

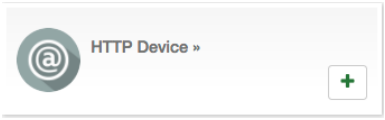


Figure 6.31: HTTP device

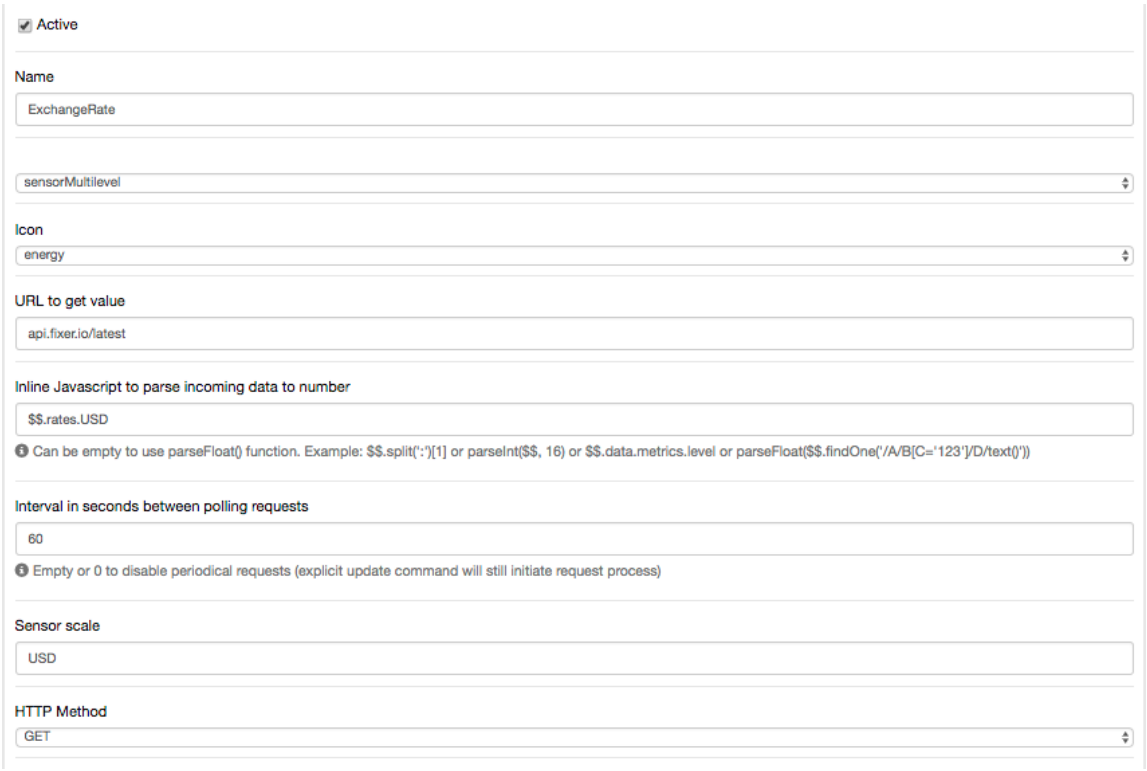


Figure 6.32: HTTP device - Configuration dialog for currency exchange "sensor"

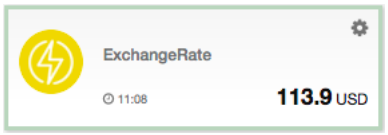


Figure 6.33: Currency Exchange Element

The website

`http://api.fixer.io/`

offers this data free of charge. Even more conveniently, the URL

`http://api.fixer.io/latest`

delivers information in a machine-readable JSON format (you can use the URL in a standard web browser to have a look at the structure.)

In the http device configuration, a multilevel sensor is chosen, since only values have to be shown and they are not only 0 and 1. Then the URL api.fixer.io is provided (attention, without http!). This will now call the whole JSON data set. For the element, the right value needs to be extracted. Here some JavaScript knowledge is needed to understand the command

"$$.rates.USD."

Finally, the refresh rate is defined and a nice name given. The other form elements are not needed here. After saving the configuration, the new element is visible. A little optimization is to show the cent value by changing the JavaScript into

"parseFloat($$.rates.USD)*100."

The sensor based on the exchange rate can now be used like any other analog sensor. Setting a trigger on certain exchange rates may be used to activate a special scene. Indeed, this is more for day traders on EUR/USD exchange, but may still be cool.

7 The Z-Wave Expert User Interface

The **Z-Wave Expert User Interface** is designed for installers, technically savvy people, and other users that know how to build and maintain a Z-Wave-based wireless network. Hence, it uses some Z-Wave specific-language and offers detailed insight into the work and data structure of the Z-Wave network. It allows users to:

- Add (include) and remove (exclude) Z-Wave devices and manage the network.
- Configure Z-Wave devices.
- Operate Z-Wave devices.
- Manage Associations between wireless devices.
- Access all data generated by the devices and perform all kind of functions and actions to the device.
- Look behind the scene into the data structures, routing mechanisms, and timings of the Z-Wave control stack. This is particularly useful for debugging and software development.

The **Z-Wave Expert User Interface** does not provide any access to a higher order business logic and automation. Please refer to the Z-Way Smart Home Interface for these functions. The user interface offers a home screen and five top menu items:

- Control: Access to functions of the wireless devices included in the network
- Device: Access to information about devices
- Configuration: Configure the devices after inclusion if needed
- Network: Add and remove devices and manage the network
- Analytics: Allows debugging the wireless network (Please note that this mean item is only shown and its corresponding functions are unlocked in the transceivers firmware

Besides the menu items, there is a configuration setting (wheel icon), a time indicator showing the time at the time zone of the gateway and a job queue indicator. Clicking on this job queue indicator opens a new tab displaying the job queue of Z-Way. Please refer to Chapter 7.8 for more information about the job queue.
All values shown in the **Z-Wave Expert User Interface** are assigned to a time stamp indicating when the value of status information was received from the wireless network. A red color of the time stamp indicates that the update request from the controller to the device for this value is pending.

7.1 Home Screen

The home screen shown in Figure 7.1 offers some high level of information about the software and a notepad where the user or installer can leave important information for future use.
The section 'Network Information' box offers some statistics about the number of devices in the network and how many of them are mains or battery-operated. The network health box will list devices that have problems:

- Low battery.
- Incomplete interview.
- Device failed.
- Inconsistencies on Association settings.

Clicking on the statement will lead to a help page explaining the problems and giving guidance for remediation.
The last infobox will contain information about devices that were removed using the device reset function. In this case, the device will leave the network but informing the controller. There is no exclusion process applied.

7.2 Control

The tab Control allows operating the various types of device and shows the reported values in case of sensors or meters. In case the control options offered here are not sufficient, please refer to Configuration » Expert Command for a full set of functions supported by the device.

7.2.1 Switch

The switch dialog shown in Figure 7.2 lists all devices of the network supporting switching, dimmer, or motor control capabilities. The device name and Z-Wave ID, as well a the current status of the switch, are given and the time of the last status update. The Update button forces an immediate update of the switch (if mains powered device). A 'Switch All' Icon shows whether or not the specific device will react to a All ON or All Off command. A green triangle indicates

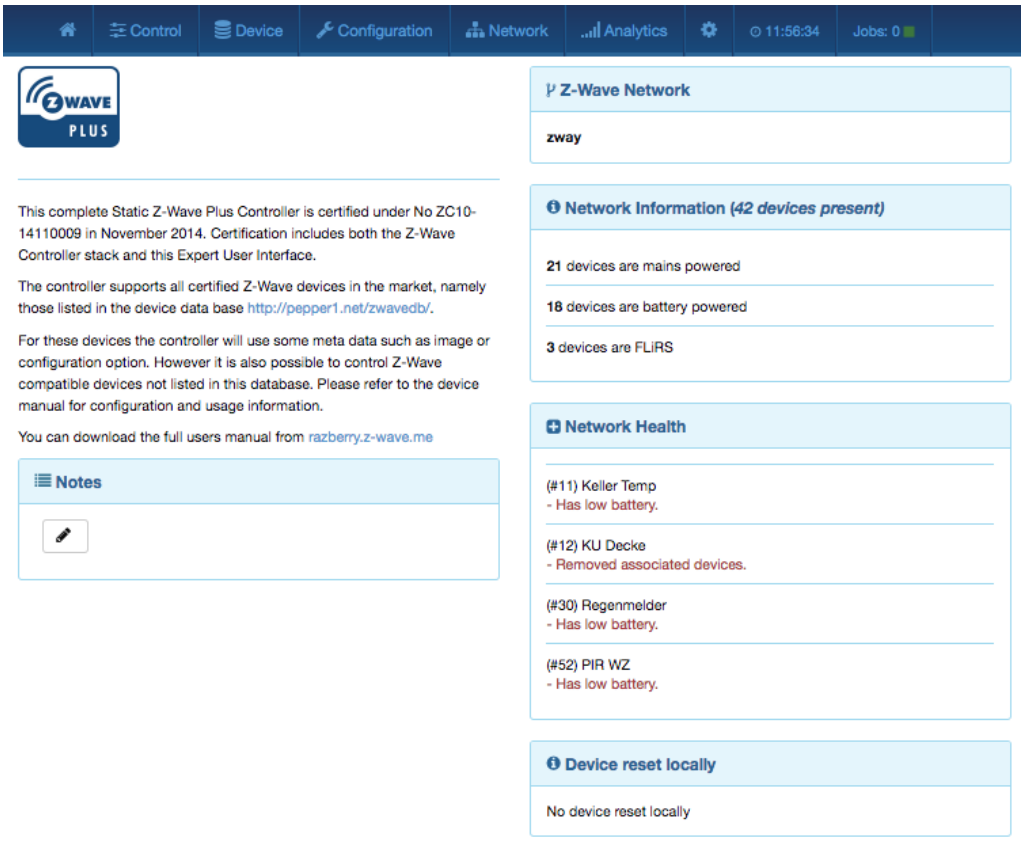


Figure 7.1: Sceenshot of the Expert User Interface Home Screen

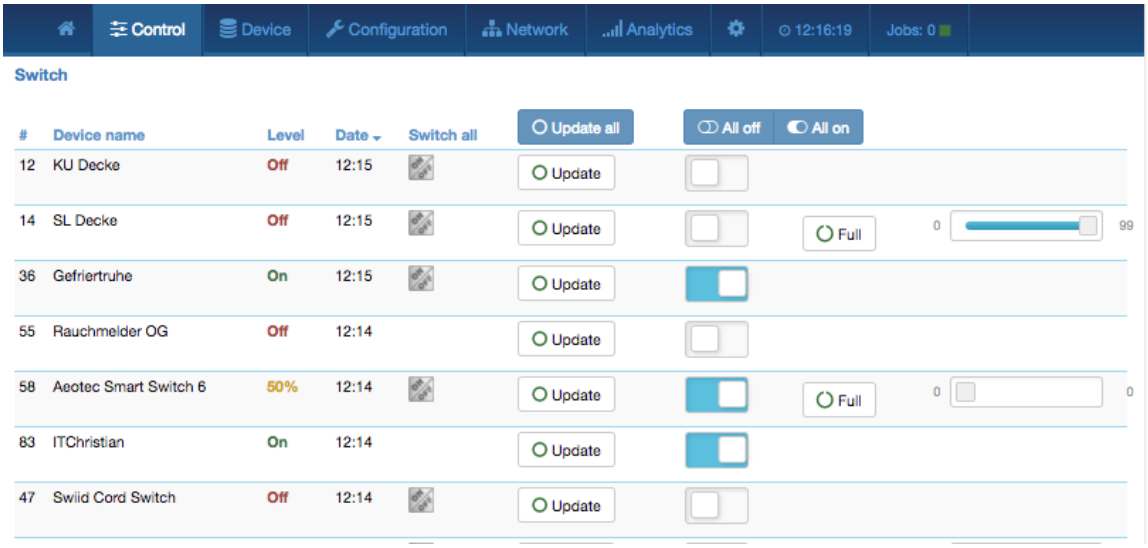
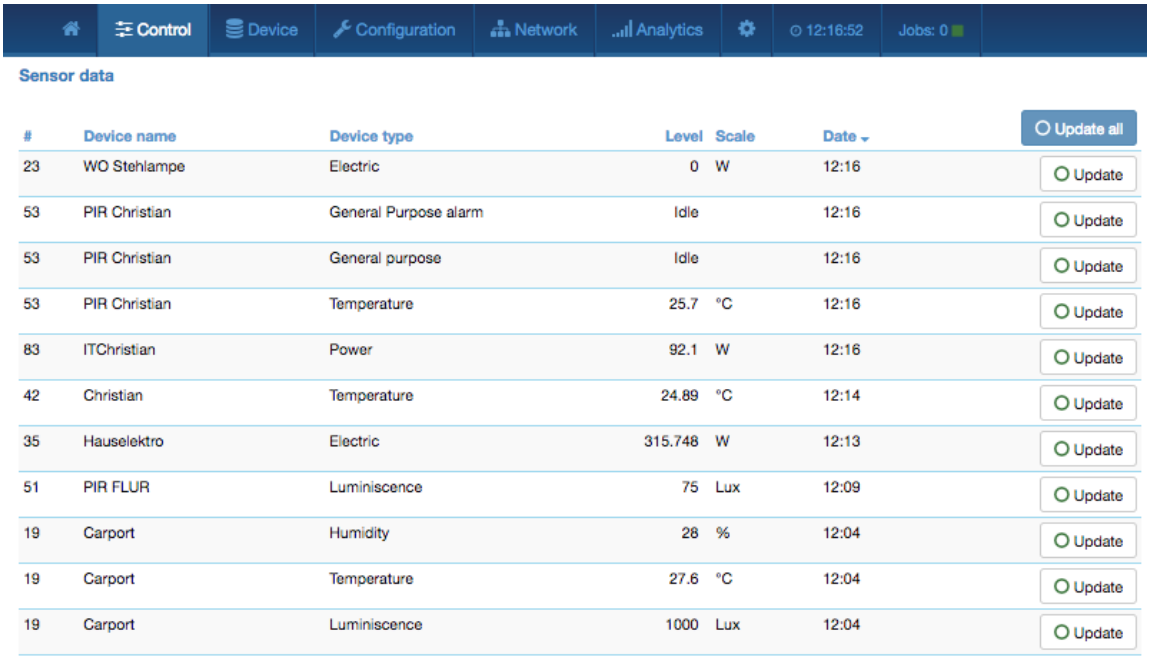
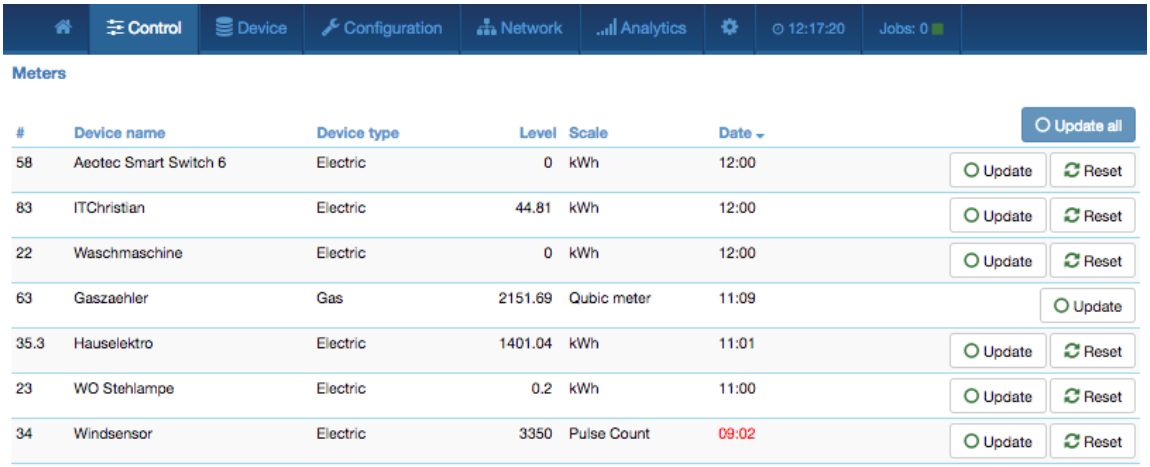


Figure 7.2: Control Interface for Switches, Dimmers and Motor Controls



#	Device name	Device type	Level	Scale	Date	
23	WO Stehlampe	Electric	0	W	12:16	Update
53	PIR Christian	General Purpose alarm	Idle		12:16	Update
53	PIR Christian	General purpose	Idle		12:16	Update
53	PIR Christian	Temperature	25.7	°C	12:16	Update
83	ITChristian	Power	92.1	W	12:16	Update
42	Christian	Temperature	24.89	°C	12:14	Update
35	Hauselektro	Electric	315.748	W	12:13	Update
51	PIR FLUR	Luminiscence	75	Lux	12:09	Update
19	Carport	Humidity	28	%	12:04	Update
19	Carport	Temperature	27.6	°C	12:04	Update
19	Carport	Luminiscence	1000	Lux	12:04	Update

Figure 7.3: Control Interface for Sensors



#	Device name	Device type	Level	Scale	Date		
58	Aeotec Smart Switch 6	Electric	0	kWh	12:00	Update	Reset
83	ITChristian	Electric	44.81	kWh	12:00	Update	Reset
22	Waschmaschine	Electric	0	kWh	12:00	Update	Reset
63	Gaszaehler	Gas	2151.69	Qubic meter	11:09	Update	
35.3	Hauselektro	Electric	1401.04	kWh	11:01	Update	Reset
23	WO Stehlampe	Electric	0.2	kWh	11:00	Update	Reset
34	Windsensor	Electric	3350	Pulse Count	09:02	Update	Reset

Figure 7.4: Control Interface for Meters

that the device will react to the command shown. All actuators can be switched on or off. Dimmer and motors controls can be operated using a slider. For dimmer, there is a button On and Full. Full turns the dimmer always to 100 %, diming value while On turns to the last dimming state before the dimmer was turned off. Clicking on the table heads reorders the table view of the data.

7.2.2 Sensors

The sensor dialog shown in Figure 7.3 lists all devices of the network providing sensor information. Device name and ID, the type of the sensor, the actual sensor value and the sensor scale is listed. The date/time column indicates when the given sensor value was received. It's possible to call for a sensor update but bear in mind that battery-operated device will only respond after the next wakeup.

7.2.3 Meters

The meter dialog shown in Figure 7.4 lists all devices of the network providing (accumulating) meter information. Device name and id, the type of the meter, the actual meter value and the meter scale is listed. The date/time column indicates when the given sensor value was received. It's possible to call for a meter update but bear in mind that

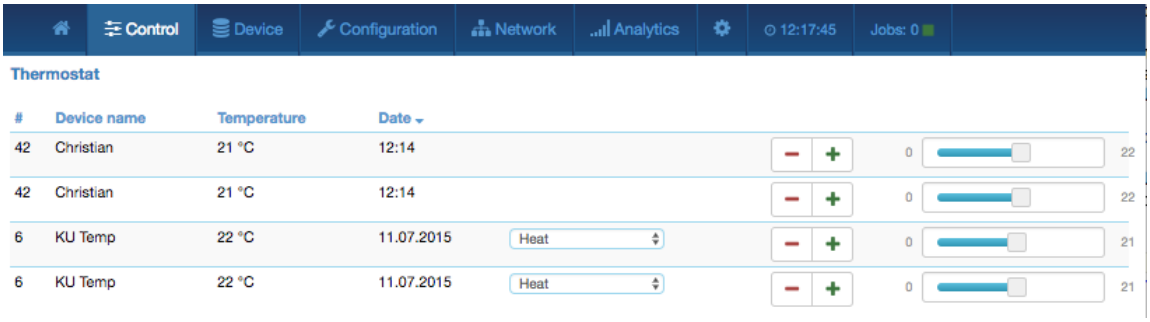


Figure 7.5: Control Interface for Thermostats

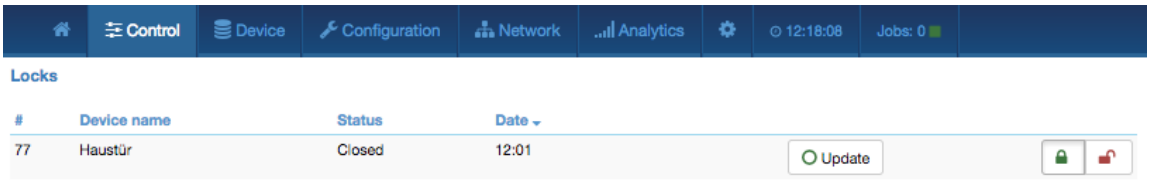


Figure 7.6: Control Interface for Locks

battery-operated device will only respond after the next wakeup. Clicking on the table heads reorders the table view of the data. A meter reset button is shown for device supporting this function.

7.2.4 Thermostats

The thermostat dialog shown in Figure 7.5 lists all thermostat devices of the network. Device name and ID and the current set point temperature is shown. The date/time column indicates when the given set point temperature was transferred to the device. The set point temperature can be changed using the + or - buttons or the slider. Clicking on the table heads reorders the table view of the data.
Some thermostats may offer different modes such as heating, cooling, off, etc. For these devices, a drop-down list shows all modes available. In this case, the setpoint is only valid for the mode selected.

7.2.5 Locks

The door lock dialog shown in Figure 7.6 lists all door lock devices of the network. Device name and ID, the current status of the lock, and the last time of the change of the status are listed. The lock can be opened or closed. Clicking on the table heads reorders the table view of the data.

7.2.6 Notifications

The notification dialog shown in Figure 7.7 lists all notification devices. Notification devices act like binary sensors, albeit offering some more capabilities. Per notification device multiple events can be reported. The notification device also allows deactivating the notification for certain events. Please note that not all devices make use of these functions.

7.3 Device

The menu Device gives access to overview pages with more detailed information about the devices in the network and their actual status.

7.3.1 Status

This dialog in Figure 7.8 shows the actual network status of all devices. All devices are listed by their node ID and name. The date/time indicates the time of the last successful communications between the controller and this device (either confirmed sending or reception). The green checkmark indicates that the device is alive. A red sign indicates

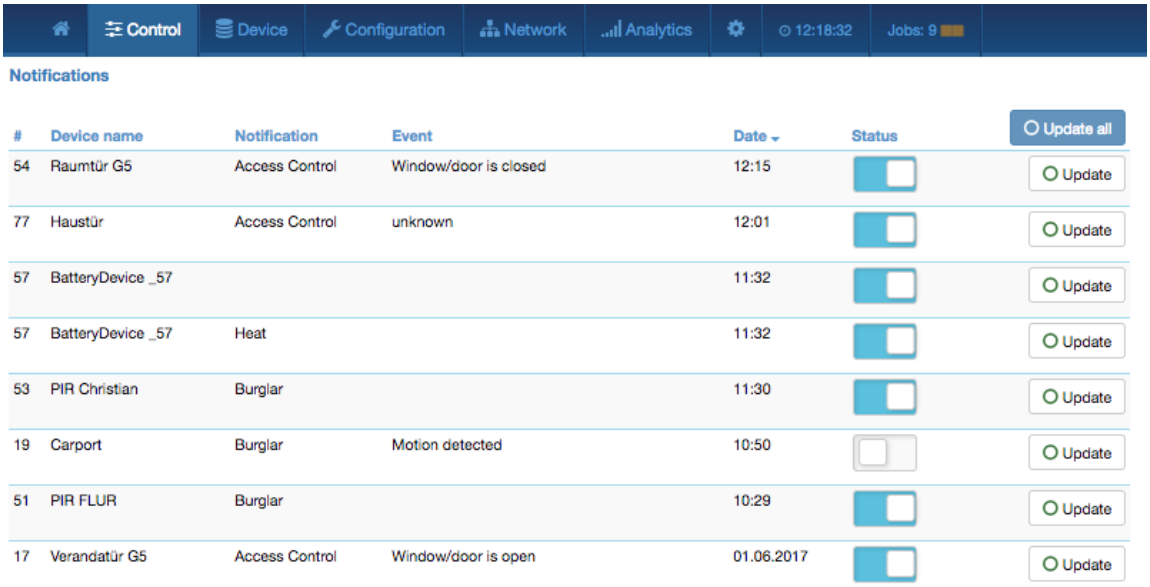


Figure 7.7: Control Interface for Notification Devices

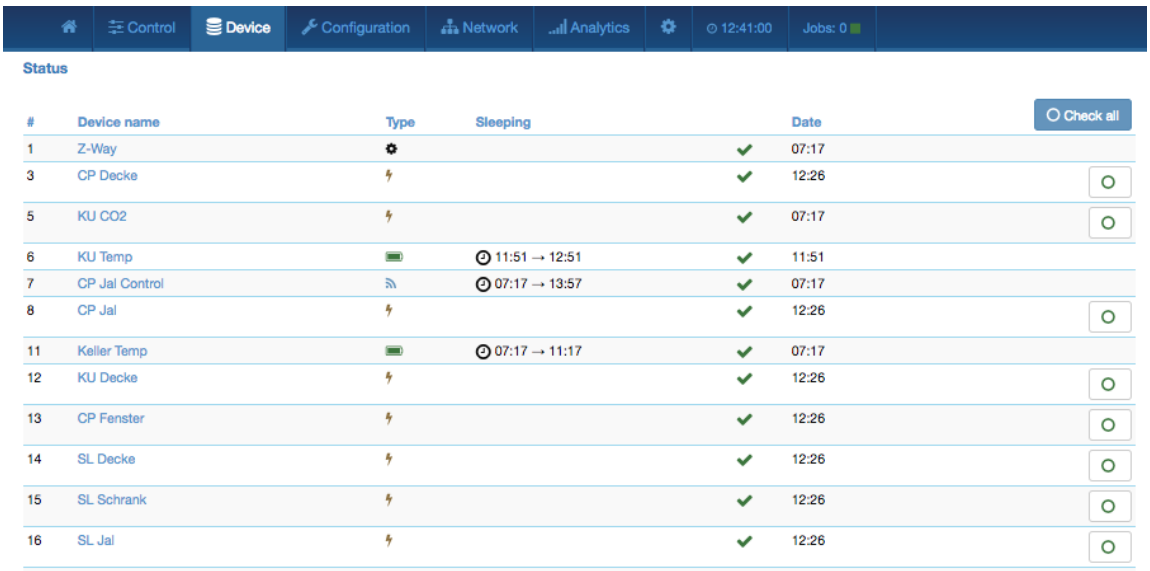


Figure 7.8: Device status overview

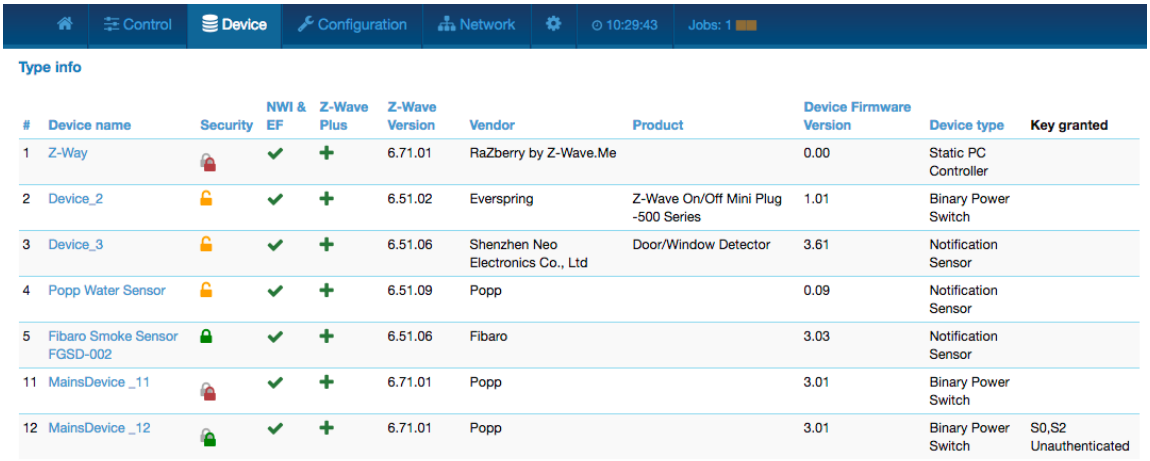


Figure 7.9: Device information overview

that the controller assumes the device not being active anymore. Mains powered devices can be checked for their network availability by hitting the button on the right-hand side.
In case the device interview and configuration were not performed properly, a little question mark icon will indicate this. Clicking on the question mark will open a window displaying the details of the interview process. The correct loading of a Device Description File[1] is indicated as well. For a battery-operated device, the time of the last wakeup, the time of the next wakeup, and the current wakeup status are shown. Clicking on the table heads reorders the table view of the data. Clicking on the table heads reorders the table view of the data.

7.3.2 Type Info

The type info dialog shown in Figure 7.9 lists all devices of the network and indicates if they support enhanced Z-Wave functions such as Security and Z-Wave Plus. Additionally, the Z-Wave protocol version, the application version and the device type indicator of the device is shown.
The security icon determines what kind of security the device supports:

- : Device does not support any security class
- : Device supports security version 1
- : Device supports security version 2
- : Device supports security version 2 but authentication failed

The last column shows the security keys granted for the device.
Clicking on the table heads reorders the table view of the data.

7.3.3 Battery

This dialog shown in Figure 7.10 gives an overview of the battery status of the battery-operated devices in the network. Devices are listed by name and id. The last reported battery level (0–100 %) including update time is shown as well as the number and type of battery if known. The Update All button will request a status update from the device. The new status will be available after the next wakeup of the device. Clicking on the table heads reorders the table view of the data.

7.3.4 Active Associations

This overview page shown in Figure 7.11 lists the current association set in the network. The Lifeline is an association to the gateway to report status changes and heard beat and can be hidden if needed. A Change button leads right to the configuration page of the device to change association settings.

[1] For more information about Device Description files, please refer to Section 7.4.1.

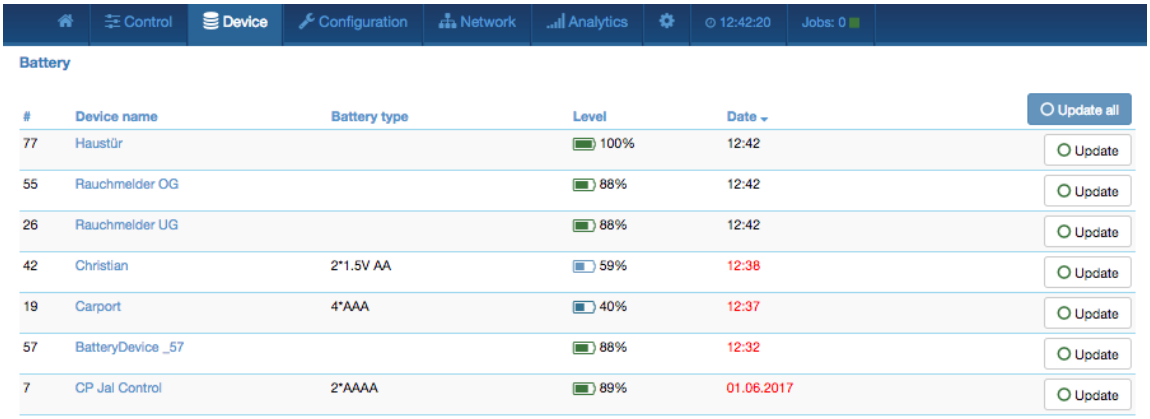


Figure 7.10: Battery status overview

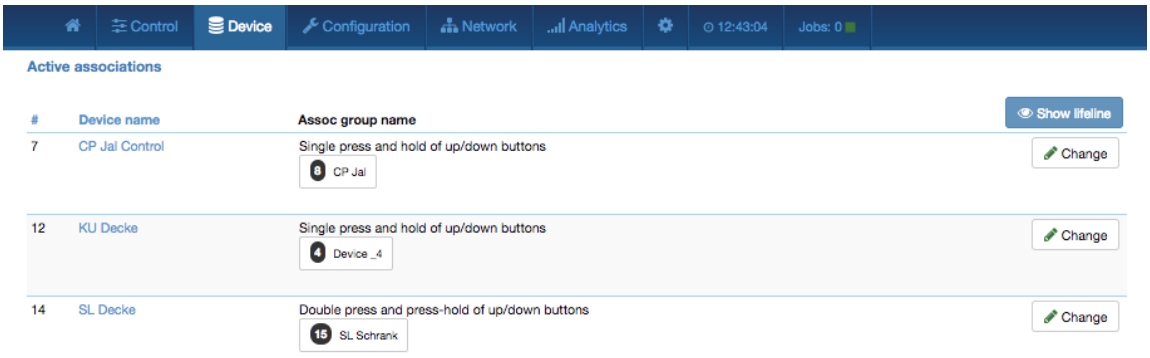


Figure 7.11: Active association overview

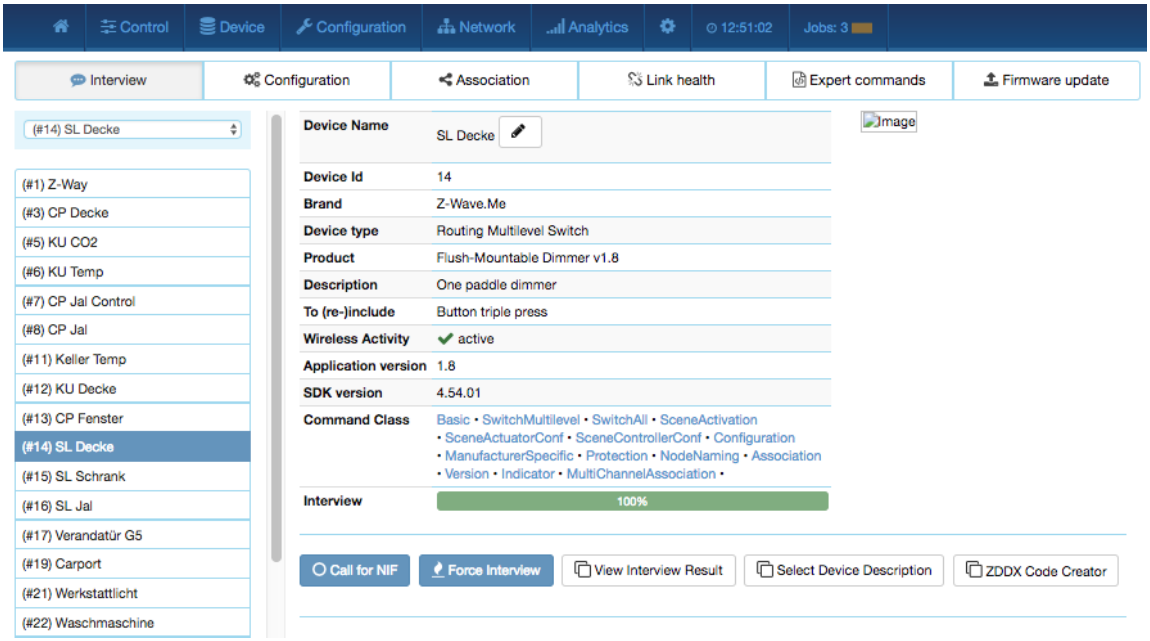


Figure 7.12: Device interview

7.4 Configuration

The tab Configuration allows configuring the functions of a particular device. Pick the device to be configured from the drop-down list or pick the device from the full list shown on the left-hand side. The functions of the device are grouped into 6 tabs.

7.4.1 Interview

Configuration › Interview, as shown in Figure 7.12, documents the result of the device interview. In this process, the controller tries to get information about the device. In case the controller finds a device description record for the device, it will display further information about the device that cannot be obtained from the device itself:

- Product Image
- Information regarding how to include the device
- Information for battery-operated devices about how to wake them up manually
- Human-readable meanings of configuration parameters and values

If the software will not automatically recognize the device and load the description record a button Select Device Description Record allows doing this manually. However, the description file must be present. Chapter 13.3 gives further information on how to create an own Device Description Record and load it into Z-Way.

The interview stage line gives information about the progress of the device interview.

There are a few reasons why an interview is not complete: In most cases the devices went to deep sleep too early to have some wireless connectivity problems. The button Force Interview allows re-doing the whole interview. The button Call for NIF requests a Node Information Frame from the device and the Button View Interview Result allows displaying the information about the different command classes found during the interview. It is also possible to force the interview of a certain command class only.

The button ZDDX Code Creator allows creating an XML description file usable for certain device databases.

The only configuration option on this tab is to change the given name of the device. During inclusion, the software generates a generic name, but it is highly recommended to change this name. The given name should be descriptive but not too long.

7.4.2 Configuration

Configuration › Configuration shown in Figure 7.13 allows configuring the device. If the specific device was recognized correctly the different configuration values are translated into human-readable dialogs. Every configuration comes with standard dialog options:

- Time Stamp, when the configuration value was last updated

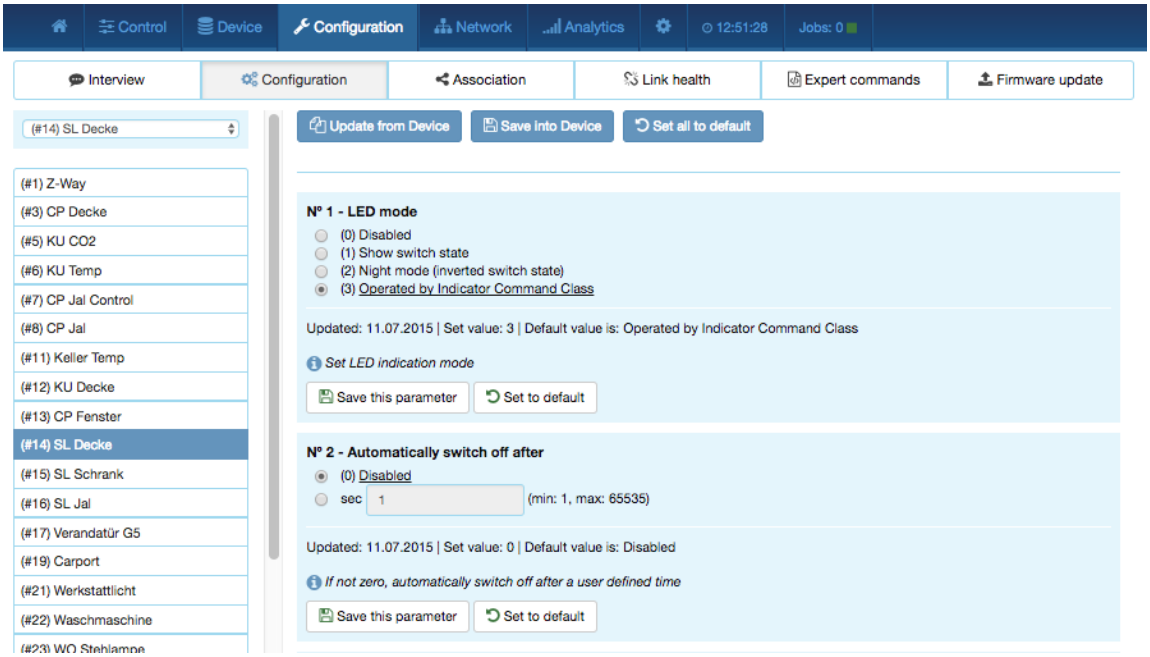


Figure 7.13: Configuration - convenient view

- Set Value as integer
- Information about the default value of this particular parameter
- Button to reset to default value given by the device itself
- Button to save the parameter into the device

If the device is not known (means there is no Device Description File assigned to it) it is still possible to set configuration values. Figure 7.14 shows the generic configuration dialog used in this case. The specific configuration parameters and its values need to be read from the device manual.

There are four more command classes that may need additional configuration and are displayed in the same dialog if the device supports them.

- Wakeup: Define the wakeup interval and the node is of the main controller taking care of the wakeup sequence. The controller will set a standard wakeup time of 1800 seconds unless the devices sets a different minimal or maximal wakeup time. In most cases the node ID of the controller is the correct setting for the target node ID and should not be changed. In case this controller is only a secondary controller, this value may change. A tool tip on the input field shows the allowed minimum and maximum wakeup time as reported by the device.
- Protection: In case the device supports local protection, meaning suppressing local use of the device, the behavior of this function can be defined. The protection command class offers more options than displayed here. Refer to the Configuration » Expert Commands tab for a complete set of controls.
- Switch All configuration: Z-Wave supports the so-called switch all function as a broadcast to all switches and dimmers. This setup defines the reaction of the device to such a Switch All command. The setting is also displayed in the Control » Switches section as little gray/green icon.

Note: For mains-powered and FLIRS devices, the button Save this parameter or Save into Device will activate the changes within few seconds. For battery-operated devices, the commands are stored to the next wakeup. It's possible and recommended to wake up the device manually to speed up the change of configuration values.

7.4.3 Association

Associations allow switching a Z-Wave Device B (target) as a result of an event in Z-Wave Device A (source). Z-Wave Device A manages a list of devices to be controlled for each event supported. The device list associated to a specific event—also called association groups—and the devices that are associated with it are shown in the association tab in Figure7.15.

In case the information is provided either by the device or by the device record stored in the software, the meaning of the events is written. Otherwise, the event group is shown unnamed as number only. In this case, refer to the devices manual for more information about the association group meaning.

The stored devices can be called from the actual device using a button. The buttons +/- used to add and remove device from the group. A dialog is opened and a device can be picked. In case this device has multiple instances, an

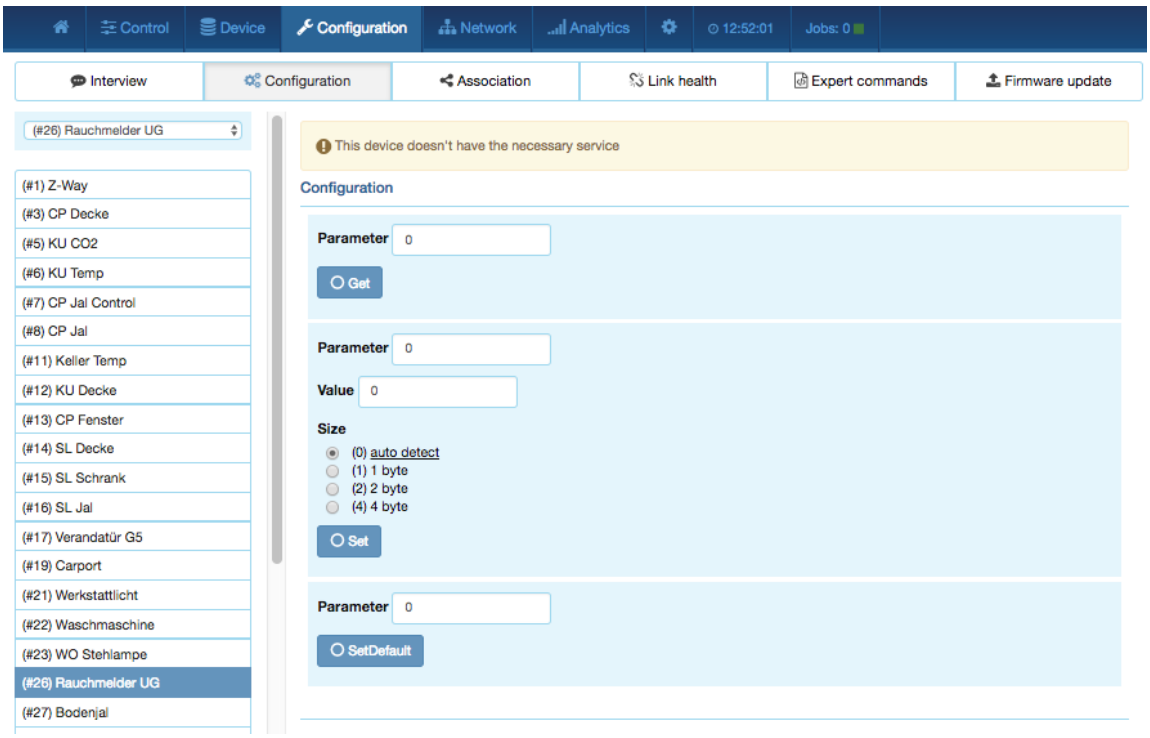


Figure 7.14: Configuration - generic view

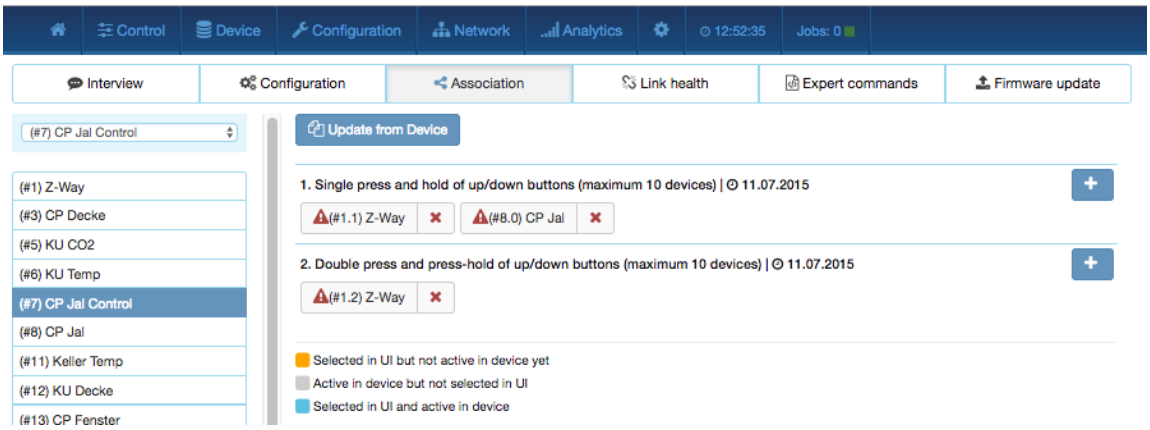


Figure 7.15: Association dialog

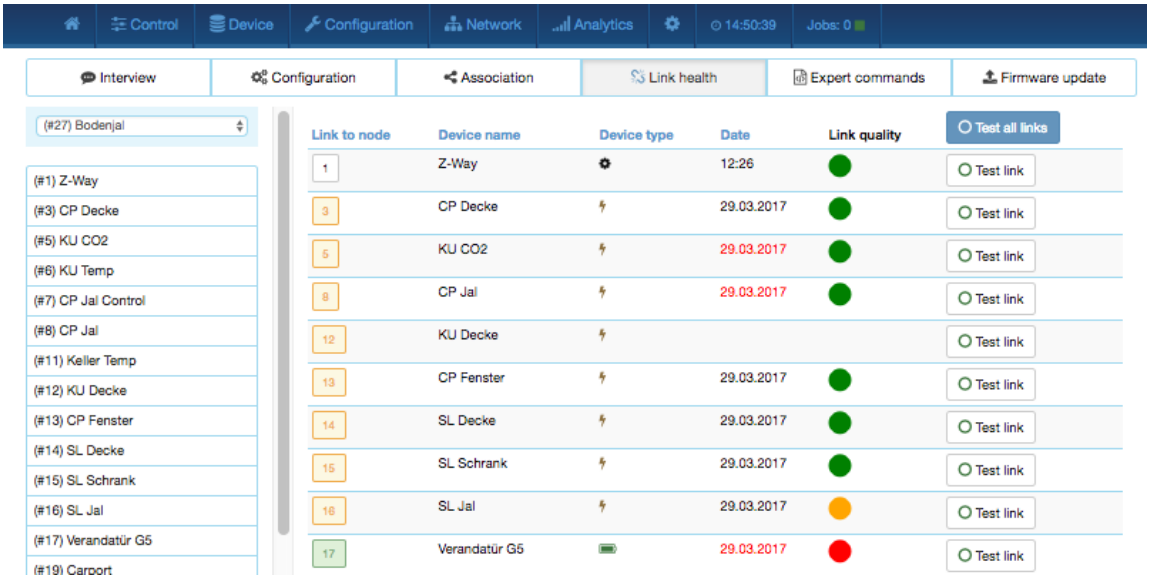


Figure 7.16: Link health

instance drop-down list will appear allowing to choose the right instance of the target device. The node ID and—if applicable—the instance ID are shown in the target device list. Move the mouse over the ID to show the complete given name of this device. The color of the device name or ID indicate the status of the association entry:

- Yellow: Selected in user interface but not stored in device.
- Grey: Active in device but not selected in user interface.
- Blue: Selected in user interface and stored in device.

7.4.4 Link Health

Modern versions of Z-Wave allow testing the quality of a link between two devices in direct wireless range. The dialog shown in Figure 7.16 lists all devices in wireless range and gives an indication about the quality of this link. The following colors are used:

- grey: untested
- green: good quality
- yellow: reasonable quality
- red: link quality insufficient

Individual links can be retested using the Test all links button. It is also possible to test all links from a given device. However, please keep in mind that this process may take several minutes to complete.

7.4.5 Expert Commands

Configuration > Expert Commands as shown in Figure 7.17 displays the status values and possible commands in a very generic way. On the left-hand side, the different instances (channels) of the device are listed in a column. In case there is only one channel (that's the case for most devices), only channel/instance 0 is shown. Clicking on the number opens a dialog showing all internal variables for the channel. The next column shows all the command classes exposed by the device. Again, clicking on the name opens a dialog with more internal status variable information for this command class. On the right-hand side, there is a list of commands. This dialog form is auto-generated from the information provided by the command class itself and not optimized for daily usage.

7.4.6 Firmware Update

In case the device supports a firmware update "over the air," this dialog is shown to perform such a firmware update. The firmware file to be uploaded must be available in a raw "BIN" or the Intel hex "HEX" file. The target field allows specifying the target memory/processor for the update process. For updating the Z-Wave firmware part a "0" must be set. The firmware updating process will take up to 10 minutes. Please don't do any other operation during this time. It may be required to activate the firmware update mode on the device to be updated. Please refer to the manual for further information about activation.

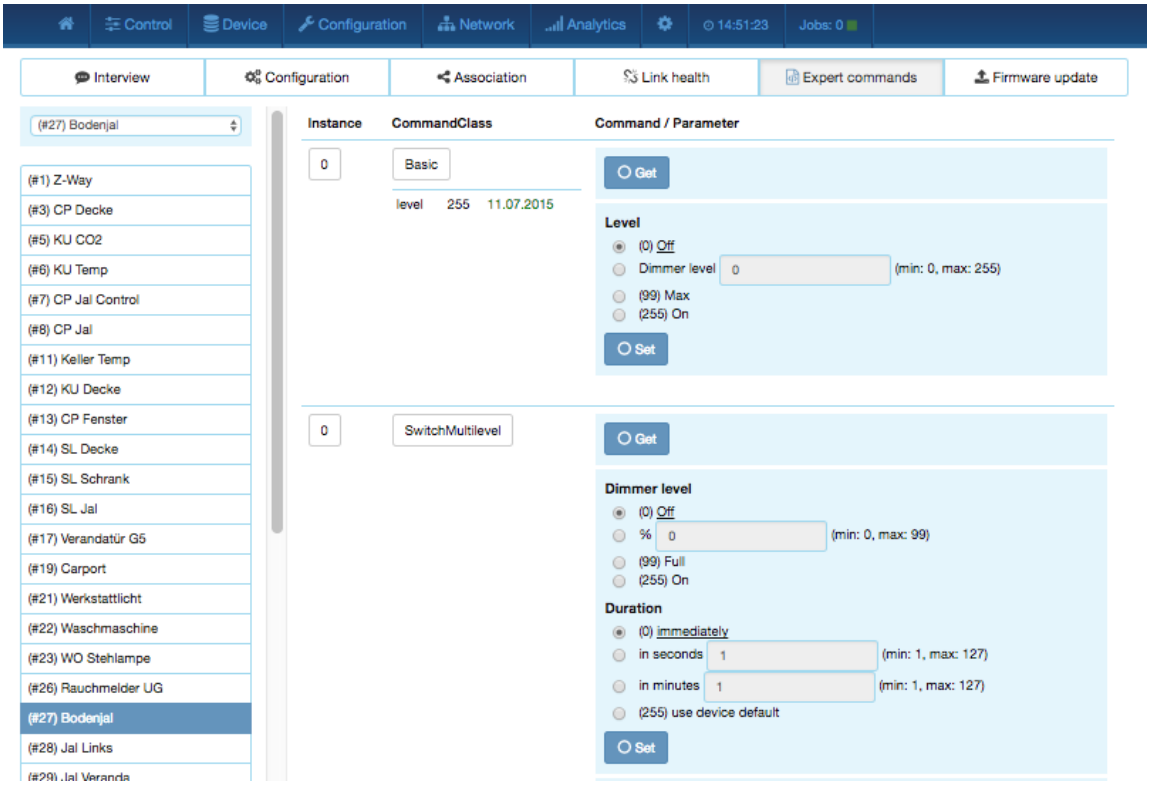


Figure 7.17: Experts commands

7.5 Network

The network section of the user interface focuses on the network as such and offers all controls and information to build, manage, and repair the wireless network of the controller.

7.5.1 Control

Network » Control summarizes all commands needed to manage the network and the controller. The page is structured in four boxes.

Device Management

The device management box as shown in Figure 7.18 allows including and excluding Z-Wave devices. A status display shows the status of the controller. The Include Device and Exclude Device button turn the controller into the Inclusion and Exclusion mode. In this case the status display changes and the resp. buttons turns into a Stop function. The inclusion and exclusion modes will time out after about 20 seconds but can always be terminated using the Stop buttons.
Another way to stop the inclusion mode is to actually include a new device. In this case the inclusion mode will stop and the node ID of the new device is shown. The controller generated a default name of the device as combination of its primary function and the new node ID. Clicking on this default name leads to the Configuration page where the name can be changed and other configuration tasks can be performed.
Please refer to the devices manual on how to do an inclusion. In case the inclusion does not work as expected, please exclude the device first. More than 90 % of all inclusion problems are caused by still included in a different network and can then not being included again.
The Exclusion mode also stops when one device was successfully excluded. This function can exclude devices from other networks too but the device need be available and functioning. To remove nonfunctioning or disappeared devices please refer to Replace Failed node or Remove Failed node.
In case the new device supports enhanced security function (Security Command Class), this controller will include the device securely. After this all data exchange between the controller and the new device is encrypted using AES encryption. For performance reasons, it mays be desired not to use the security function. The slider Force unsecure inclusion turns the controller into a mode where all security functions are suppressed for the included device. Security functions of other devices are not impacted. In case the new device supports Security S2 right after inclusion, a pop-up window

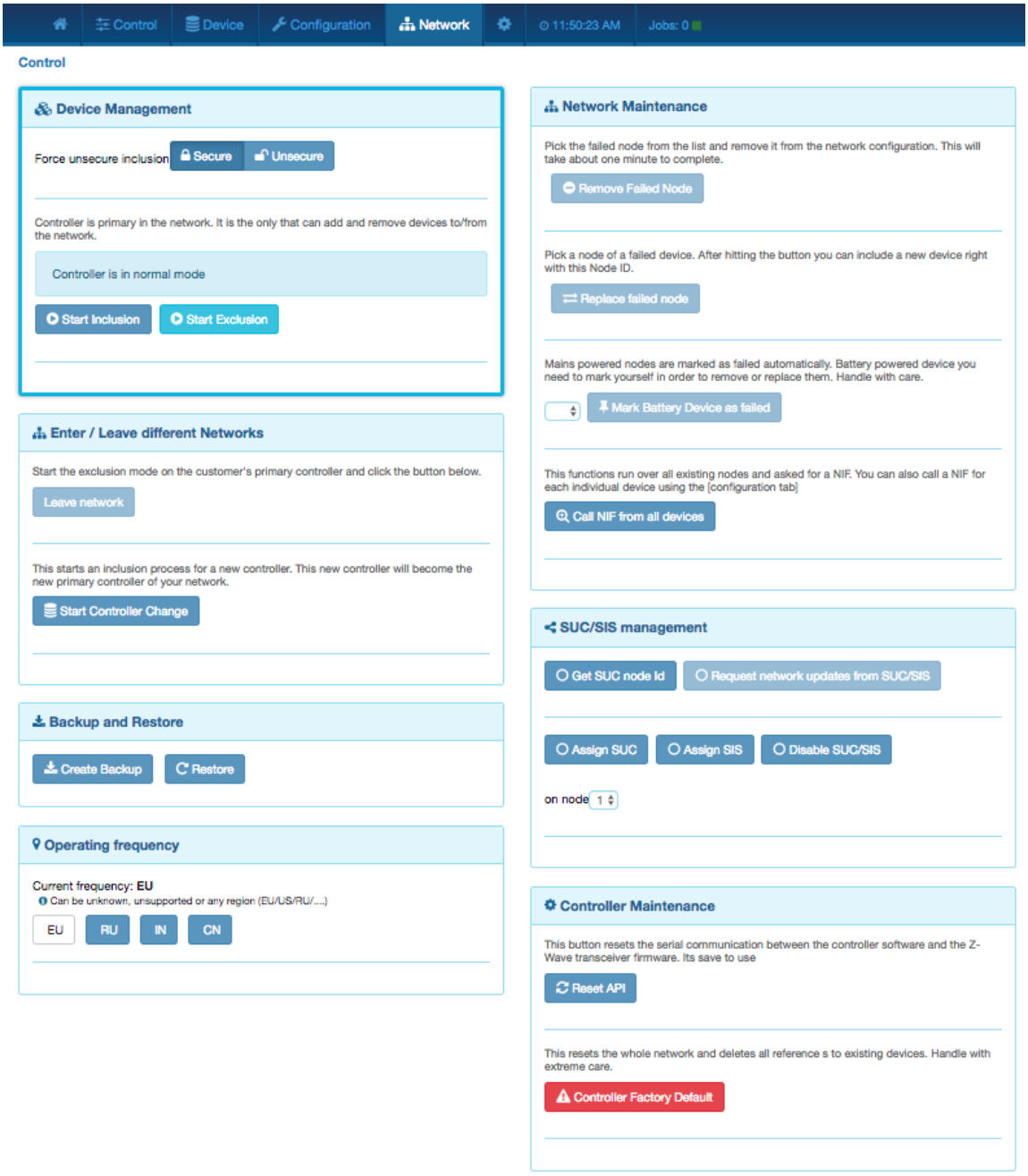


Figure 7.18: Network Management

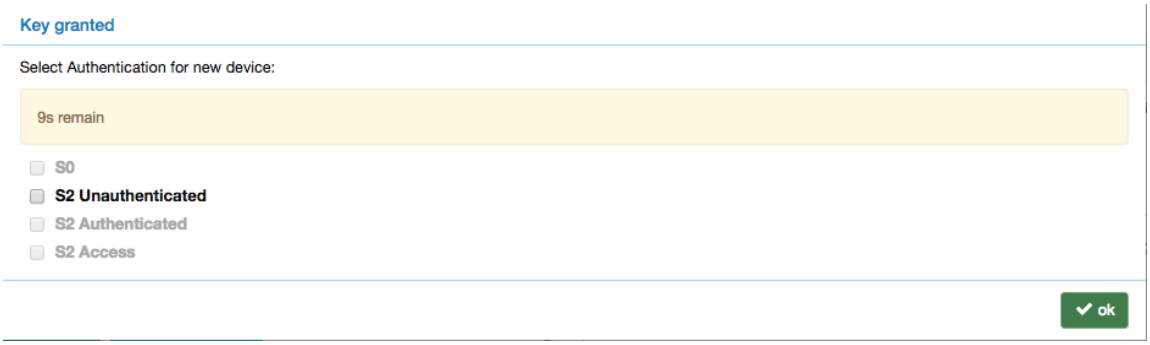


Figure 7.19: **Z-Wave Expert User Interface** - S2 key selection

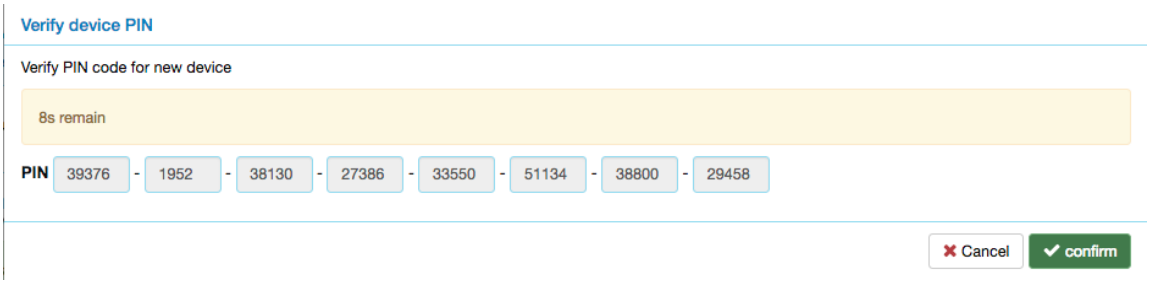


Figure 7.20: **Z-Wave Expert User Interface** - S2 key display

will appear asking the user to grant the S2 security keys. A few basics about this process: Security in Z-Wave means among others that all communication between two nodes over the air is encrypted. Encryption however requires a key to encrypt and all the device communicating shall have the very same key—usually called network key. Z-Wave Security 2 handles 4 different network key. They differ by their level of trust.

- S0:The network key is needed to communicate with a device capable of Security CC Version 1. Hence, this is more for backward compatibility.
- S2 Unauthenticated: The device-specific key of the device included is not verified. In case the device has its key (as PIN number of QR Code) on the enclosure, it is possible to compare the two numbers but the key assignment does not verify this choice.
- S2 Authenticated: Right after inclusion, the device-specific key (as a PIN or AR code) must be manually provided to the included controller in order to verify the identity of the device just included.
- S2 Access Control: This key requires similar authentication than S2 authenticated. This additional key is provided to separate control of door locks and other access devices from other devices.

Each device will request one of more network keys according to the device usage and implementation idea. The default key is likely S2 Authenticated but Door Locks will ask for S2 Access control and small devices without external label request S2 Unauthenticated only. Figure 7.19 shows the dialog to grant the keys. The requested key is displayed in bold letters. The user has 20 seconds to select the keys to be granted. If no selection is made with 20 seconds, the process will time out and the device is included unsecure. Please note that some devices may still offer valid functions while other will deny any function outside a secure environment. It is recommended to grant the keys requested, but there may be certain environments where other selections may make sense.

Once the selection is made before timeout a second window will request the authentication of the device. In case of S0 or S2 Unauthenticated, the window displays the device-specific key for information only as shown in Figure 7.20. If a key is visible on the device, it is recommended to compare the numbers. However, the key is granted without any further interaction by the user.

In case S2 Authenticated or S2 Access Control keys were requested the dialog windows asks for providing the Device information by the user. This can be done by typing in the 5-digit PIN code (first 5 numbers of the device-specific key) as shown in Figure 7.21 or by scanning the QR code on the device shown in Figure 7.25.

In case of Access or Authentication, the user must insert either a PIN number or scan a QR code from the device just included. This insures 100% that the device just included is really the device in hand. If authentication fails, an error message is displayed. It is not possible to redo the key authentication only. You must exclude and re-include the device.

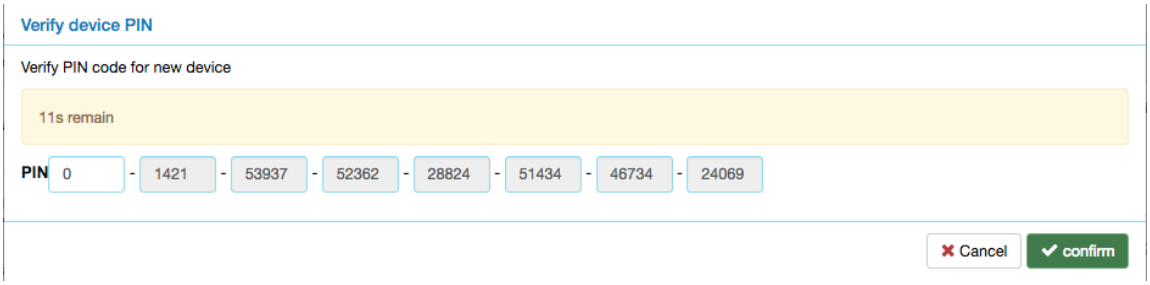


Figure 7.21: Expert User Interface - S2 authentication

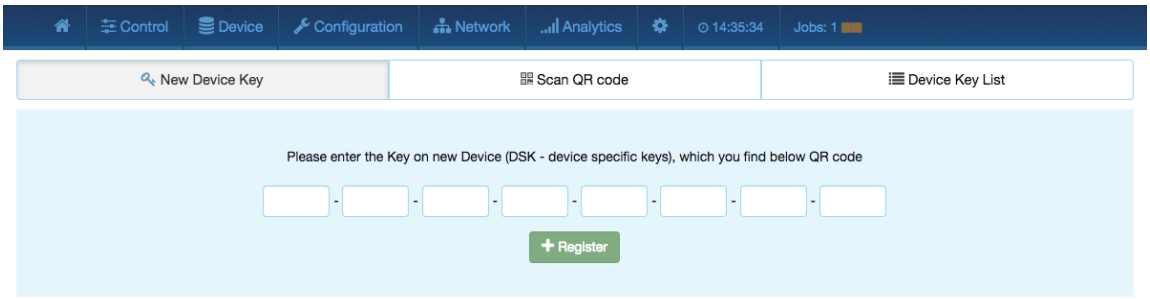


Figure 7.22: Smart Start - enter Device Key (DSK)

Smart Start

Smart Start is a new way to include devices into Z-Wave using the QR code provided with S2 authentication. The user scans the QR code thats is stored in an internal so called provisioning list. Smart Start Devices will then announce to be included when powered up. In case the S2 key is in the provisioning list the controller will automatically include this new device without any further user interaction.

Smart Start is controlled by the button Activate/Deactivate Smart Start. The button Smart Start opens a submenu allowing to register Device keys either by typing the 8 groups of numbers as shown in Figure 7.22 or using a QR code scanner as shown in Figure 7.23. Please note that a standard web browser running on a standard PC may not provide the capability to scan QR codes.

Figure 7.24 finally shows the lost of Device keys already registered in the system and not used. Used means in this context that the corresponding device was powered up in proximity of the controller and got included automatically. Please note that the Razberry Shield or UZB or any hardware used must provide a firmware with SDK >=6.81. Otherwise Smart Start will not work.

Enter/Leave different Network

This button will only be active if Z-Way is in factory default state. Only in this case the controller can be added to a different network as secondary controller. The controller of the other network must be in either the Inclusion or in the Exclusion mode, and the Learn button confirm the process.

In case the new primary controller supports Security S2 a dialog window will pop up right after finishing the inclusion by the new controller. This dialog window will show **the PIN number and the QR Code of this Z-Way controller**

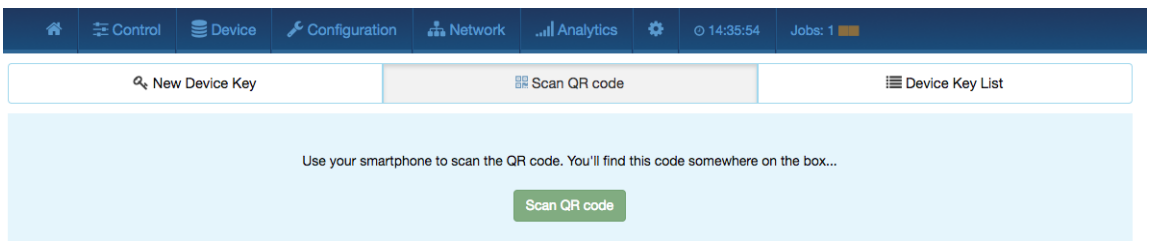


Figure 7.23: Smart Start - scan QR code (on smart phones only

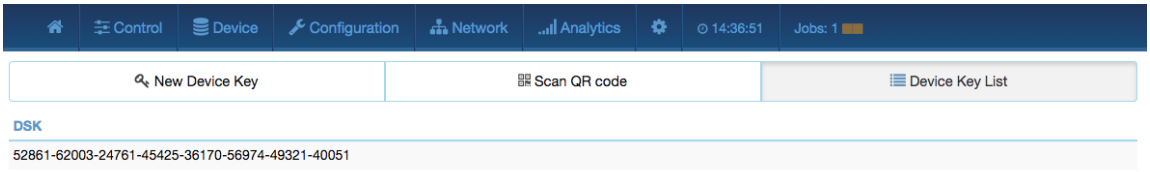


Figure 7.24: Smart Start Provisioning list

zws2dsk:34028-23669-20938-46346-33746-07431-56821-14553

Figure 7.25: Z-Way- own key for authentication

needed to authenticate against the new controller.
The Start Change Controller function actively hands over the role of the primary controller of the network to a controller that will be included using the normal inclusion process. The controller status dialog shows the mode and its termination.

Backup and Restore

The next dialog box of the page allows creating and using a backup. The backup file is stored on the local computer. Please note that any restore will overwrite the existing network. The restore operation must therefore be confirmed in another message box. A checkbox defines if the node information in the Z-Wave chip itself will be overwritten as well. This operation result in a possible loss of all network relationships and may require a re-inclusion of devices. Handle with care!

Controller Maintenance

The controller maintenance offers two reset buttons. The Z-Wave Chip Reboot restarts the operating system of the Z-Wave transceiver chip. This can be done safely all the time.
The Reset Controller turns the controller back into factory reset. All connections to included devices and all configurations and settings are lost. This function must be handles with extreme case. An additional dialog requires to explicitly confirm the function. Only use this function if you know what you do!

Operating Frequency

This dialog allows changing the operating frequency. Please note that a wrong frequency will block all Z-Wave traffic and make the device inoperable. Frequencies can only be changed within one frequency group. These are the frequencies shown side by side (e.g. EU, RU, IN, ...). Changing to a frequency outside this group will technically work but the wireless range will be few centimeters only. Hence, this for workbench testing only. The device will reboot after a frequency change so please allow some time for restart.

Network Maintenance

The function Remove Failed Node allows removing a node that is no longer communicating with the controller. After multiple failed communications with a device the controller will mark this device as failed and avoid further communication. This function finally removes such a device from the network configuration. The drop-down list will only show IDs of failed nodes. If this list is empty this is a good sign!
It is also possible to Replace a Failed node with a new node. This is a combination of removing the failed node and adding a new node. Using this function makes sure the next included node has the same node ID as the failed node. The drop-

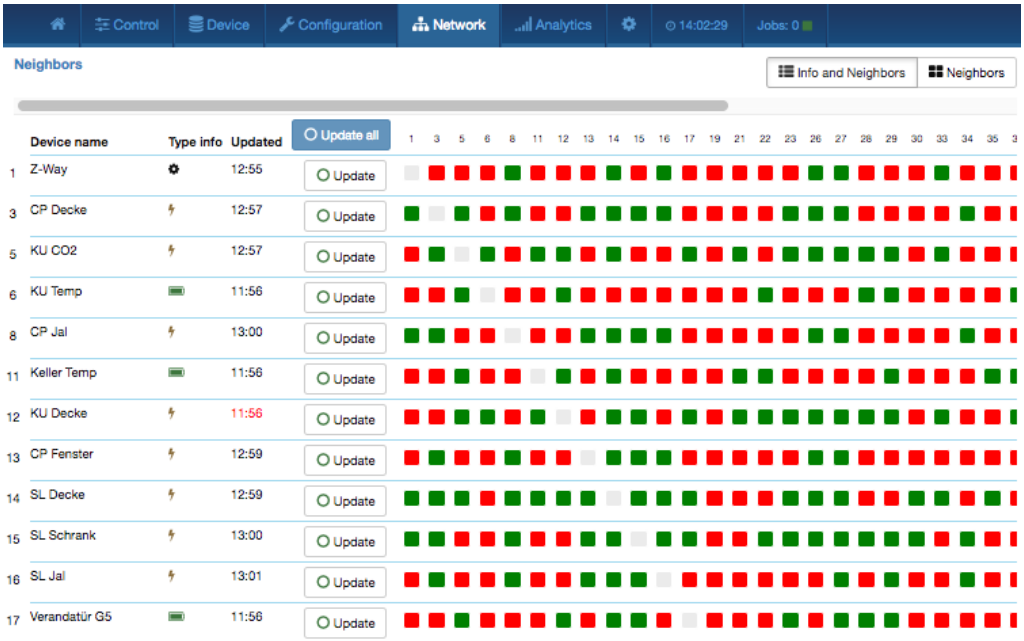


Figure 7.26: Neighbors

down list will only show IDs of failed nodes. If this list is empty this is a good sign! Battery-operated devices are mainly in deep-sleep state and will not answer to communication requests. Hence, the controller will never automatically detect if a certain device is defect or gone. The function Mark battery device as failed manually marks battery-operated devices as failed so that they can be removed or replaced. The drop-down list shows all battery-operated devices but this does not mean that they are failed.
The Request NIF from all devices function is just a convenient way to retrieve a Node Information Frame from all devices of the network.

SUC/SIS Management

The SUC/SIS Management pane allows manipulating the self-organization of the Z-Wave network. Don't use this unless you are a developer who knows when and why this is needed for testing purposes.

7.5.2 Neighbors

This table in Figure 7.26 shows the neighborhood relationship of devices. The id, the name and the type of the node are listed. Green indicates that the two devices are in direct wireless neighborhoods and don't need any other device to forward their signals. A red color between two nodes indicated that routing is needed between these devices. The Update button calls the device to scan its neighborhood and report back the result to update its own line of the routing table.

7.5.3 Reorganization

The reorganization page controls as shown in Figure 7.27 an algorithm that reorganizes the network relationships and fixes problems. With checkboxes various stages of the algorithm can be selected. The result of the reorganization is shown in a log and can be downloaded. The network reorganization calls for every node to redetect its neighbors. This operation will work if there is a working route to this device and this device is not sleeping. If the operation fails, the algorithm will have three more attempts to reflect possible routes to the very device that may be reestablished during the reorganization. Detection of neighbors for battery-operated devices will be started after all mains-operated devices are processed. The requests to battery-operated devices will be queued. For more information about job queuing, please refer to Chapter 7.8.

7.5.4 Network Map

The **Poltorak-Chart** as a way to map the network is an extremely powerful, informative but the same time very complex viewgraph of the network situation in a home. The chart visualizes the possible links between the nodes and

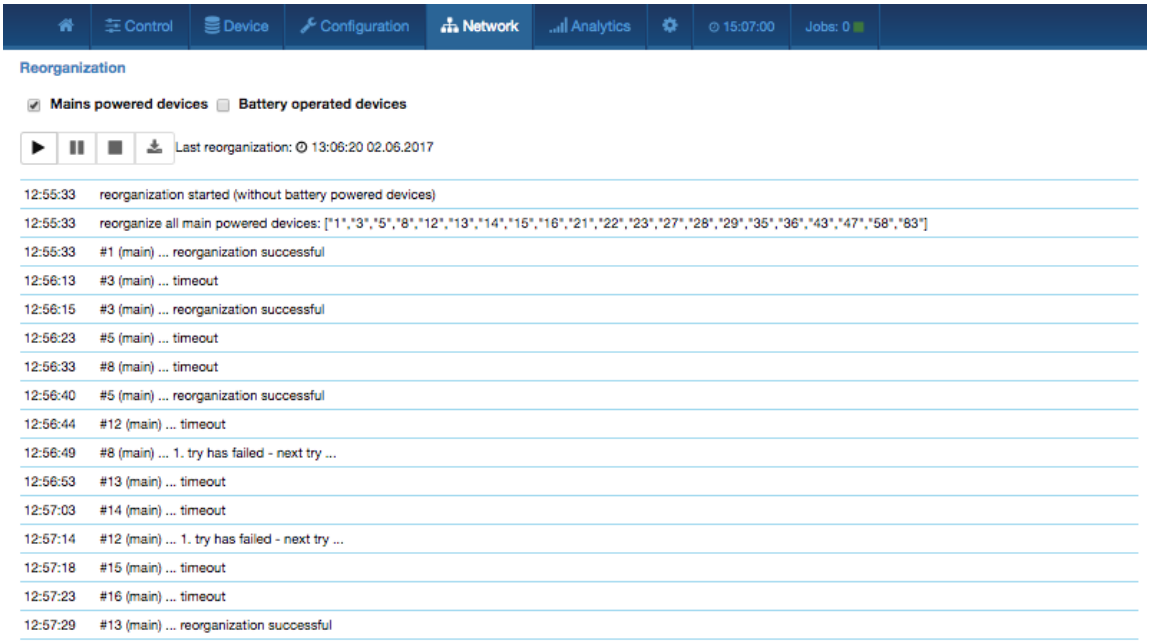


Figure 7.27: Reorganization

how they are used. If provided by the devices the chart will furthermore show complete routes and the signal strength of the individual links of a route. However only devices with SDK greater or equal to 6.71 will provide this additional information.

Initially all nodes are displayed with equal distances. It is possible to drag and drop the nodes to match the distances between them with the real distances. This will always work quite well if the Z-Wave network is in one floor only. A 2D map can be uploaded as background image to support the mapping. If the network is distributed on multiple floors it is recommended to do a best guess to keep the round initial view.

Figure 7.28 shows a typical chart. By clicking on a certain node and then moving the mouse over it again, it is possible to analyse the traffic from and to this very node. This allows focusing on the situation of this node only.

The lines between the nodes represent the wireless connections and the communication between the nodes. The following information is encoded in these lines:

- Color: The color indicates the wireless signal strength of the connection if it can be measured. Red means a very high wireless signal. The device is likely very closed or in direct sight of the each other. A black color means a standard wireless strength, gray indicates that the received signal strength (RSSI value) is not known.
- Thickness: The Thickness of the line indicates the amount of traffic running over this line. This can be direct communication between the two links or routed traffic. If a route exist there will be a single pixel line. Every line thicker than a pixel shows real traffic.
- Dotted versus solid: A dotted line indicates that this link is sometimes just not working.

7.5.5 Timing Info

The Timing tab in Figure 7.29 shows some very valuable timing information of communications between the controller and other devices. All other devices the controller has communicated with are shown in a list. The number of packets and the percentage of successful communication are shown. This can give an indication about the stability of the communication link between the controller and this device. On the right-hand side, the timing delay of each communication is shown and color-coded. A red number indicates that this communication finally failed. A communication without rerouting attempts as shown as green and a rerouting attempt is coded in black. The fact that a communication failed (red) may indicate that there is a severe problem in the network or in the device. It is however also possible that a battery-operated device just went back to sleep too fast. Z-Wave professionals can read a lot out of this timing information particularly when combined with the routing table. Please refer to Chapter 8 for more details on troubleshooting Z-Wave networks.

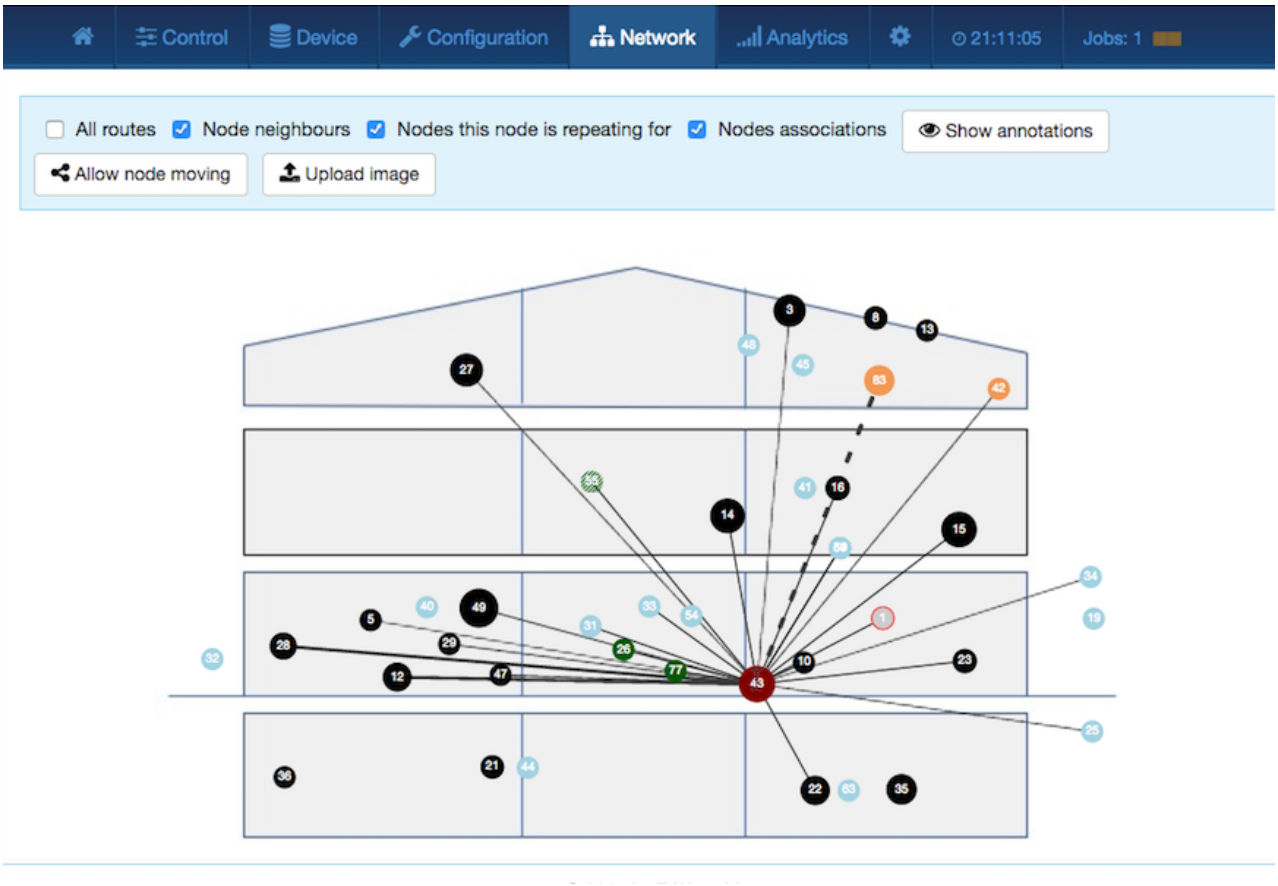
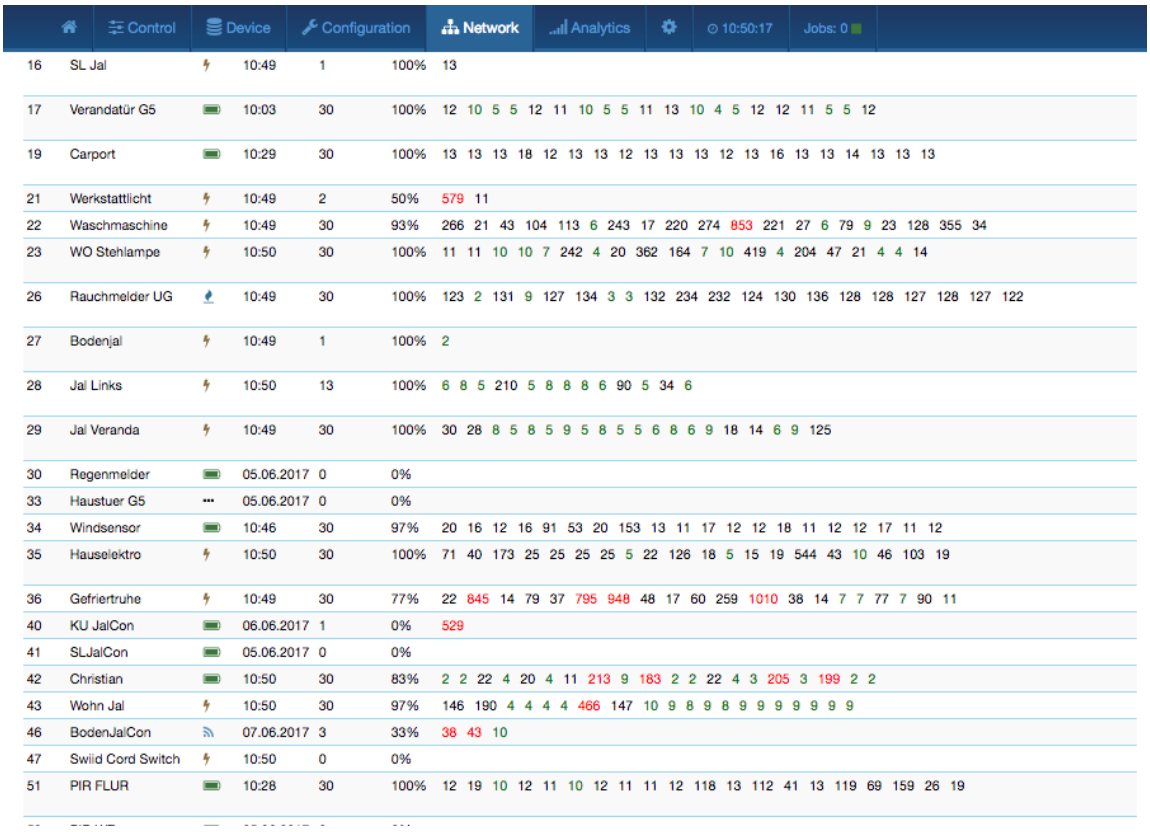


Figure 7.28: Poltorak-Chart



16	SL Jal		10:49	1	100%	13
17	Verandatür G5		10:03	30	100%	12 10 5 5 12 11 10 5 5 11 13 10 4 5 12 12 11 5 5 12
19	Carport		10:29	30	100%	13 13 13 18 12 13 13 12 13 13 13 12 13 16 13 13 14 13 13 13
21	Werkstattlicht		10:49	2	50%	579 11
22	Waschmaschine		10:49	30	93%	266 21 43 104 113 6 243 17 220 274 853 221 27 6 79 9 23 128 355 34
23	WO Stehlampe		10:50	30	100%	11 11 10 10 7 242 4 20 362 164 7 10 419 4 204 47 21 4 4 14
26	Rauchmelder UG		10:49	30	100%	123 2 131 9 127 134 3 3 132 234 232 124 130 136 128 128 127 128 127 122
27	Bodenjal		10:49	1	100%	2
28	Jal Links		10:50	13	100%	6 8 5 210 5 8 8 8 6 90 5 34 6
29	Jal Veranda		10:49	30	100%	30 28 8 5 8 5 9 5 8 5 5 6 8 6 9 18 14 6 9 125
30	Regenmelder		05.06.2017	0	0%	
33	Haustuer G5		05.06.2017	0	0%	
34	Windsensor		10:46	30	97%	20 16 12 16 91 53 20 153 13 11 17 12 12 18 11 12 12 17 11 12
35	Hauselektro		10:50	30	100%	71 40 173 25 25 25 25 5 22 126 18 5 15 19 544 43 10 46 103 19
36	Gefriertruhe		10:49	30	77%	22 845 14 79 37 795 948 48 17 60 259 1010 38 14 7 7 77 7 90 11
40	KU JalCon		06.06.2017	1	0%	529
41	SLJalCon		05.06.2017	0	0%	
42	Christian		10:50	30	83%	2 2 22 4 20 4 11 213 9 183 2 2 22 4 3 205 3 199 2 2
43	Wohn Jal		10:50	30	97%	146 190 4 4 4 4 466 147 10 9 8 9 8 9 9 9 9 9 9
46	BodenJalCon		07.06.2017	3	33%	38 43 10
47	Swiid Cord Switch		10:50	0	0%	
51	PIR FLUR		10:28	30	100%	12 19 10 12 11 10 12 11 11 12 118 13 112 41 13 119 69 159 26 19

Figure 7.29: Timing Info

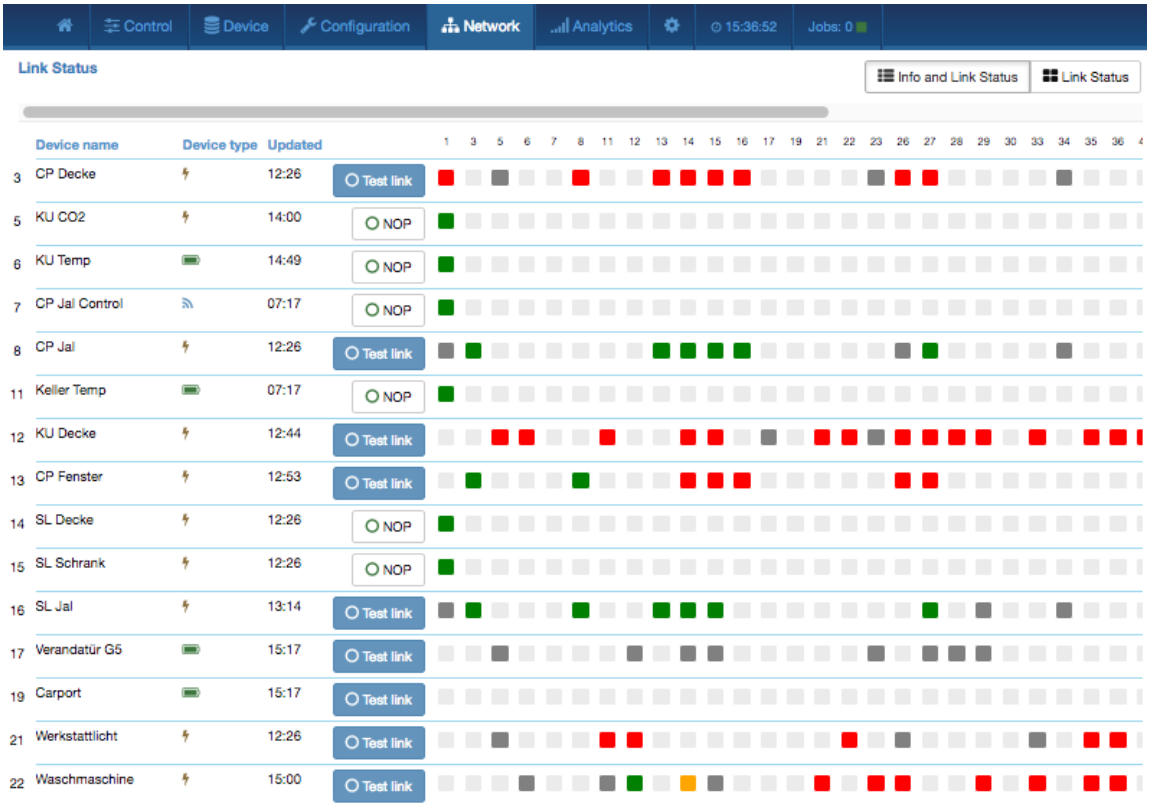


Figure 7.30: Link Status

7.5.6 Link Status

The link status map, as shown in Figure 7.30, summarizes the device-specific link status information from the configuration sections.

The following colors are used:

- grey: untested
- green: good quality
- yellow: reasonable quality
- red: link quality insufficient

The links from one device can be retested using the Test link button. However, please keep in mind that this process may take several minutes to complete. For devices that do not offer a link status function in their firmware, there is a simple connection test to the Z-Way controller using a "NOP." Please note that this function does not test a direct wireless connection but the route to the controller only.

7.5.7 Controller Info

This menu item as shown in Figure 7.31 provides some internal and very technical information about the Z-Way controller software, hardware, and firmware. The different submenu items are self-explaining.

Some buttons allow special maintenance functions:

- View Job Queue : This button opens a new tab with a list of all wireless jobs in the system. For more information about this, please refer to Chapter 7.8. This is needed for debugging purposes only.
- Send Controller NIF : Sends out the Node Information Frame of the Z-Way controller. This is needed for debugging purposes only.
- Debug Mode : When active this button is shown in green. Active debugging unlocks some special function embedded in the rest of the Expert User Interface. Among them is the user interface to edit so-called "Postfix Records." The postfix function allows changing device capabilities after the device interview. Typical postfix entries suppress certain functions that are announced by the Node Information Frame of certain command classes but not or wrongly implemented. Postfix is also used to rename certain functions to more meaningful terms. Post fix entries are typically created during device testing.
- Show controller data and Show controller's device data : The controller is a special node in the network but still a node. Therefore, buttons allow accessing the device specific data of the controller and issue a Node Information Frame.

Figure 7.31: Controller Info

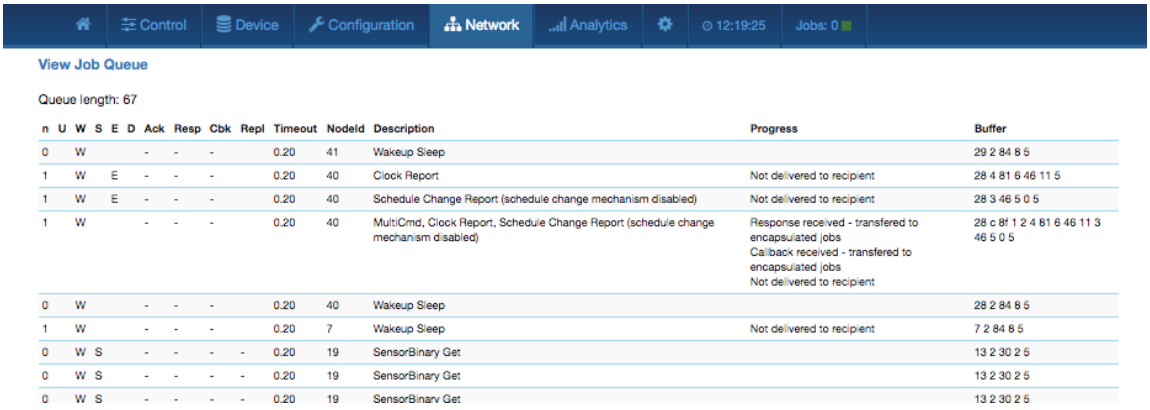


Figure 7.32: Job Queue

A third button gives access to the controller specific data. Most of these data is only relevant to developers.

- Firmware Update : This function allows updating the function of the Z-Wave transceiver used by Z-Way. Only valid update files will be offered. It is possible to add so-called tokens. These is a special string used to identify special function firmware or beta firmware. They are sued to make sure that this special firmware is only available to those that really need, e.g. for testing.

The last block of the dialog shows the availability of the function calls on the serial API between the Z-Wave transceiver and Z-Way. "Not implemented" means that Z-Way is either not knowing about the function indicated by the function ID or does not make any use of it. A red function call ID or name indicated that Z-Way can use this function, but the transceiver's firmware did not report to offer this service.

Annex C gives a full overview of the Functions used and supported by Z-Way.

7.5.8 Basic Set Handling

Z-Wave protocol defines a minimal level of interoperability via Basic Command Class. Most devices do act on `Basic Set` commands mapping it to `Switch Binary Set` or `Thermostat Mode Set` or some other. Z-Way is able to control other devices using speical Command Classes or using Basic Set.

Z-Way is handling incoming `Basic Set` and creates a virtual device that reflects the state of the last `Basic Set` command recieved. This allows to work with old devices that do not send `Reports` or `Central Scene Notification` to the LifeLine association group. Z-Way is ignoring `Basic Get` commands.

7.5.9 Security Considerations

Z-Way supports both Security S2 and Security S0 protocols and implements very strict security rules. See Table 7.1 for more information.

7.6 Analytics

The analytics menu offers functions to troubleshoot a Z-Wave network. Details about the dialogs are provided in Chapter 8.

7.7 Setup

The setup dialog offers various options to adapt the behavior of the **Z-Wave Expert User Interface** :

- Language: Pick your user interface language by clicking on one of the flags.
- System Settings: This option allows setting the date format and the time zone.
- Report Problem: This option allows reporting user interface bugs. Please note that the form will transmit the test, he option email address for questions and answers plus some internal version and status information.

Negociated security	Unsecure NIF	Security S0 NIF	Security S2 NIF
No or failed security	ZWavePlusInfo Version MultiCmd Security SecurityS2 Supervision ManufacturerSpecific NodeNaming ApplicationStatus CRC16 PowerLevel Association AssociationGroupInformation MultiChannelAssociation DeviceResetLocally TransportService InclusionController	-	-
Security S0	ZWavePlusInfo MultiCmd Security SecurityS2 Supervision NodeNaming ApplicationStatus CRC16 PowerLevel AssociationGroupInformation DeviceResetLocally TransportService InclusionController	Version MultiCmd ManufacturerSpecific NodeNaming ApplicationStatus CRC16 PowerLevel Association AssociationGroupInformation MultiChannelAssociation DeviceResetLocally InclusionController	-
Security S2	ZWavePlusInfo SecurityS2 Supervision TransportService	-	Version MultiCmd ManufacturerSpecific NodeNaming ApplicationStatus CRC16 PowerLevel Association AssociationGroupInformation MultiChannelAssociation DeviceResetLocally InclusionController

Table 7.1: Supported Command Classes depending on the security level negociated

7.8 Job Queue

The job queue, as shown in Figure 7.32, offers some deep insight into the dynamics of the controller software. It visualizes the (wireless) task execution of the system. Every communication attempt of the controller is queued and then handed over to the Z-Wave chip for execution. The list shows the jobs pending and the jobs that are completed or failed.

A legend informs about the meaning of the different flags n, U, D, E, S, and W. The timeout value counts back from 20 seconds once the job was sent. Even when it is completed, the job will stay in the queue marked as done (D) for some more time to allow inspection. The target node ID, a description of the communication message, information about the process, and the real bytes of the message are shown as well.

8 Troubleshoot the Z-Wave Network

The **Z-Wave Expert User Interface** is perfectly suited to troubleshoot networks and find and fix problems. Troubleshooting a Z-Wave network works along the lines of the communication stack.
Problems can occur on the radio layer, the networking layer, and the application layer. To identify and fix problems, it makes sense to work bottom up through the network stack issues.
Most of the troubleshooting functions are accessible on the menu item Analytics. However, this menu item will only be displayed if the firmware on the Z-Wave chip supports some special functions needed for troubleshooting purpose.

8.1 Radio Layer

Problems on the radio layer come from interference and noise generated by defect or nonconforming electrical gear causing electromagnetic emissions (baby monitor, old cordless phones, wireless speakers, motors, etc.). Other Z-Wave networks with unusual high traffic can also be a root cause of problems. It is also possible that certain other wireless networking services (first and foremost cellular network G4 routers or base stations, also called LTE) may cause interference if they are too closed to the Z-Wave network.
The menu item Analytics 〉 Background Noise offers a view chart displaying the background noise on the two communication channels used by Z-Wave. Channel 1 refers to the 9.6 Kbit/s and 40 kbit/s communication modes, channel 2 points to the 100kbit/s data rate. Figure 8.1 shows this viewgraph. There is an obvious floor of noise with some other "needles." This noise floor—in Figure 8.1 at about -85 dBm for channel 1 and -90 dBm for channel 2—is the minimum level a Z-Wave transceivers signal must surpass in order to be decoded by the Z-Wave receiver.
The lower the noise level the better the wireless situation. Noise levels below - 95 dBm are very good, levels above -70 dBm are very bad.
Please note that this noise level is measured right on the controllers location or wherever the hardware running Z-Way is positioned. It may make sense to move the measuring device around to see the noise level at different locations. Since the Analytics 〉 Background Noise viewgraph is only updated once per minute, you may want to use the tool Analytics 〉 Noise Gauge, as shown in Figure 8.2. In this case, the display is updated every two seconds.
If the noise floor is too high, you need to find the source of the noise. The device running Z-Way can be used as mobile device too, thanks to the built-in Wi-Fi. In this case, it needs to be powered with a power bank as shown in Figure 8.3.
Walking around with the Noise Gauge enabled may help to track down the jamming device. The closer the controller hardware gets to the source of the noise, the higher the background noise level will be.
The "needles" above the noise floor show communication from other Z-Wave networks around. Having this is not a real problem unless other networks generate heavy traffic. A rule of thumb is that there should not be more than 30 % of the time allocated by traffic of other Z-Wave networks. If there is more traffic, there will be a need to troubleshoot the other Z-Wave network first. The chart Analytics 〉 Network Statistics, as shown in Figure 8.4, shows a ratio of own traffic versus traffic seen from other networks.

8.2 Network Layer - Devices

Devices can have two faulty states:

- They are dead, removed, faulty, stolen, etc. In case of a mains-operated device, the central Z-Way controller will eventually find out that the device is not responding. It will put the device in the failed node list (for more information about failed node please refer to Chapter 7). The Devices 〉 Status Overview as shown in Figure 8.5 indicates if a device is failed or not. It is possible to make a test if the device is working.
- The device is working but constantly sending unsolicited messages. This is a rare but not impossible behavior. The simplest way to find out is to consider the packet sniffer. Figure 8.6 shows the Analytis 〉 Sniffer View of the **Z-Wave Expert User Interface** .

Another option to detect faulty devices is the Network 〉 Timing Info View.
Figure 8.7 shows this view. The timing information lists one entry for every communication between the controller and the device. The number refers to the time (in x * 10 ms) the message took before being confirmed; the color gives a rough indication of what happened:

- Green: Successful communication with device in direct wireless range.
- Black: Successful communication with device using a route.

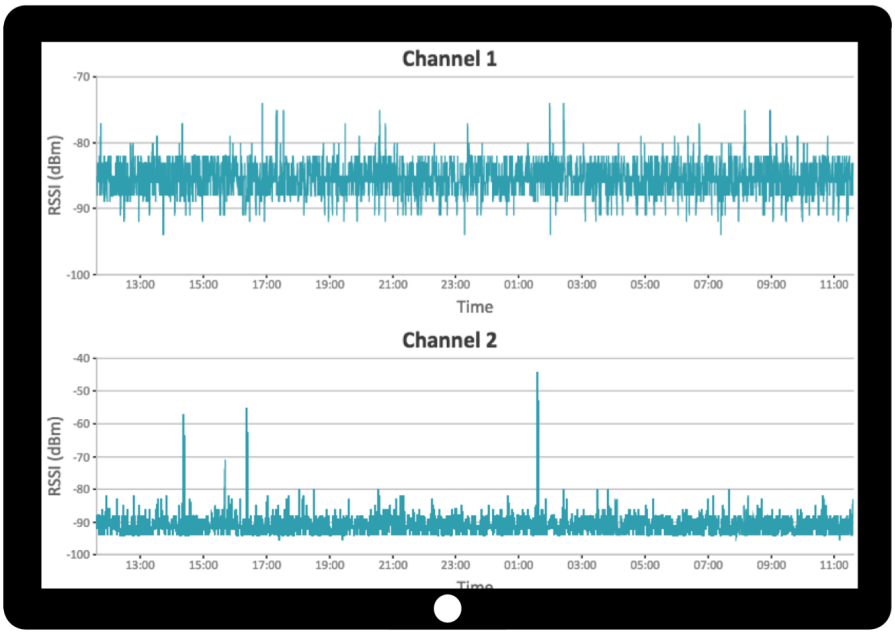


Figure 8.1: Background Noise

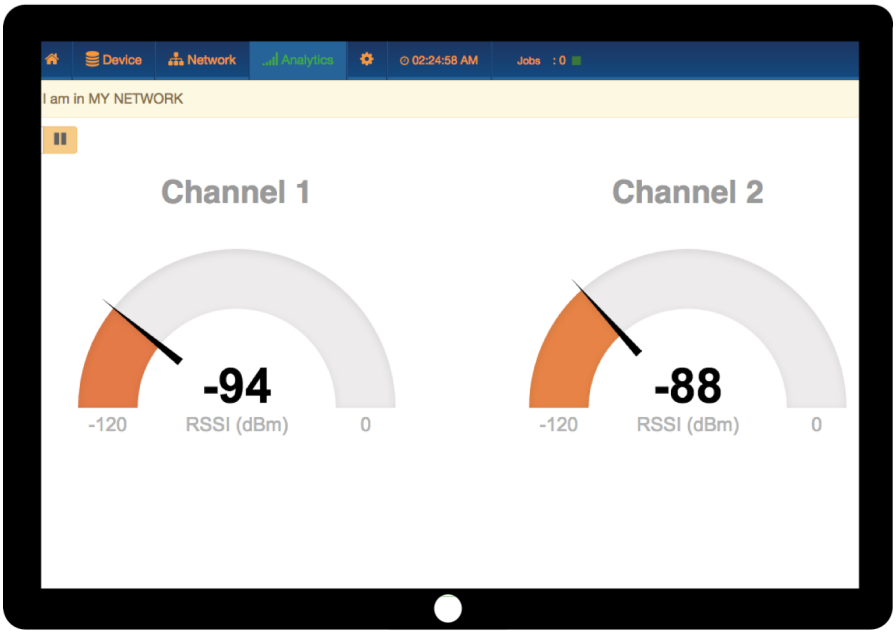


Figure 8.2: Realtime Measurement of Background-Noise

Figure 8.3: Powerbank to power the Z-Way controller for mobile use

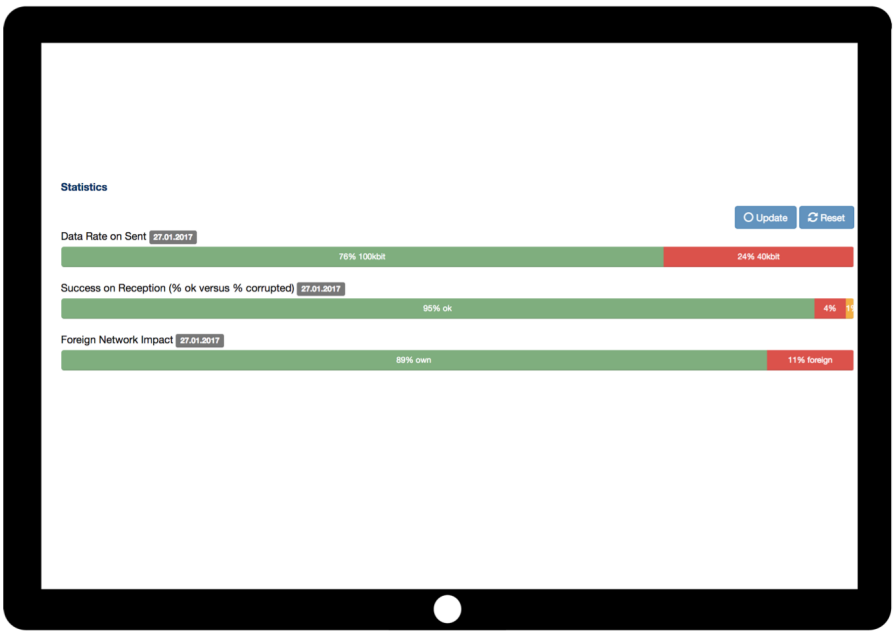


Figure 8.4: Network Statistics Display

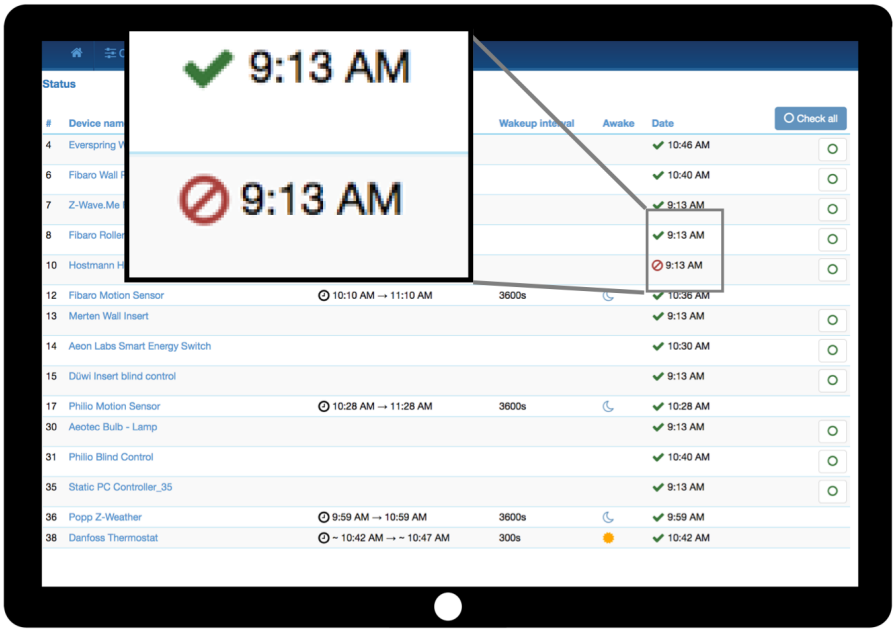


Figure 8.5: Status Page Z-Way

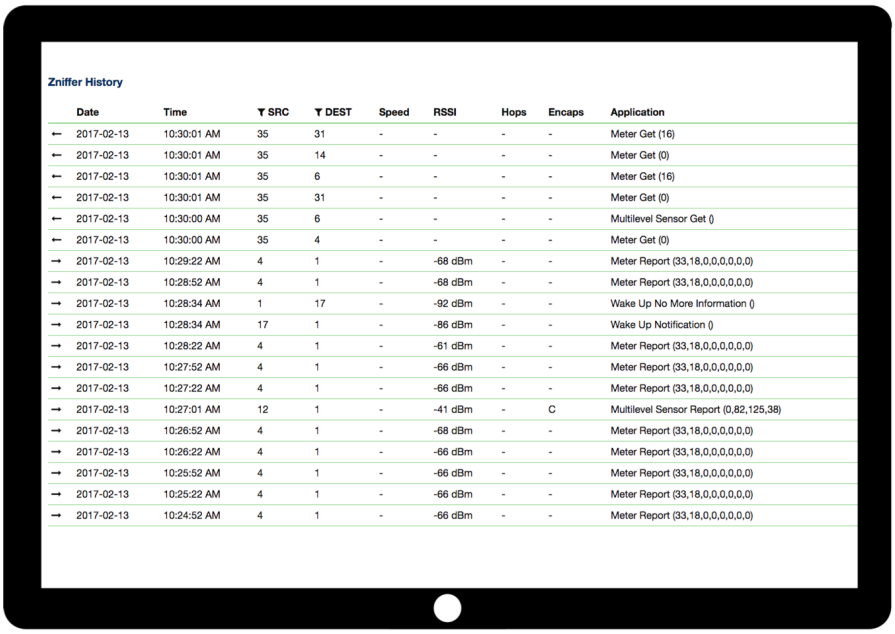
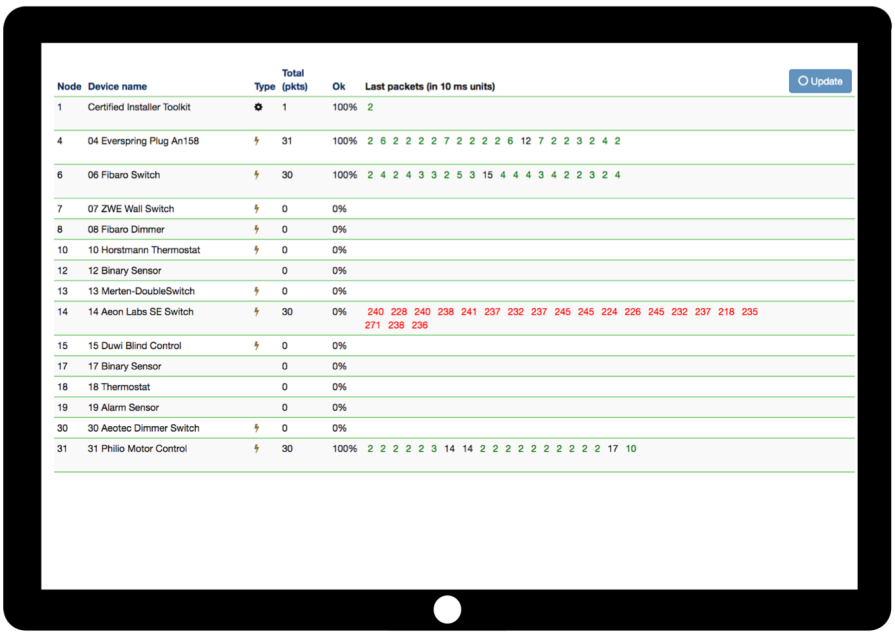


Figure 8.6: Packet Sniffer



Node	Device name	Type	Total (pkts)	Ok	Last packets (in 10 ms units)
1	Certified Installer Toolkit		1	100%	2
4	04 Everspring Plug An158		31	100%	2 6 2 2 2 2 7 2 2 2 2 6 12 7 2 2 3 2 4 2
6	06 Fibaro Switch		30	100%	2 4 2 4 3 3 2 5 3 15 4 4 4 3 4 2 2 3 2 4
7	07 ZWE Wall Switch		0	0%	
8	08 Fibaro Dimmer		0	0%	
10	10 Horstmann Thermostat		0	0%	
12	12 Binary Sensor		0	0%	
13	13 Merten-DoubleSwitch		0	0%	
14	14 Aeon Labs SE Switch		30	0%	240 228 240 238 241 237 232 237 245 245 224 226 245 232 237 218 235 271 238 236
15	15 Duwi Blind Control		0	0%	
17	17 Binary Sensor		0	0%	
18	18 Thermostat		0	0%	
19	19 Alarm Sensor		0	0%	
30	30 Aeotec Dimmer Switch		0	0%	
31	31 Philio Motor Control		30	100%	2 2 2 2 2 3 14 14 2 2 2 2 2 2 2 2 2 2 17 10

Update

Figure 8.7: Paket timing of a fresh Z-Wave network

- Red: Failed communication (after a total of nine attempts).

Figure 8.7 shows the situation in a network just installed. It can be seen that there is only communication with few devices, e.g. no polling of sensors, etc. While this is not a problem, the chart shows that devices 4, 6, and 31 are in direct range and all communication works perfectly well (green, low number). Device 14 seems to be a real problem child. The controller tries all the time to reach this node but always fails. At some point in time, the controller will accept that node 31 is dead and put him into the "failed node list."

Figure 8.8 shows a network that is a bit more complex, has more communication and is aged. Again node 20 is a defect device that just needs to be replaced. The following interesting patterns can be seen:

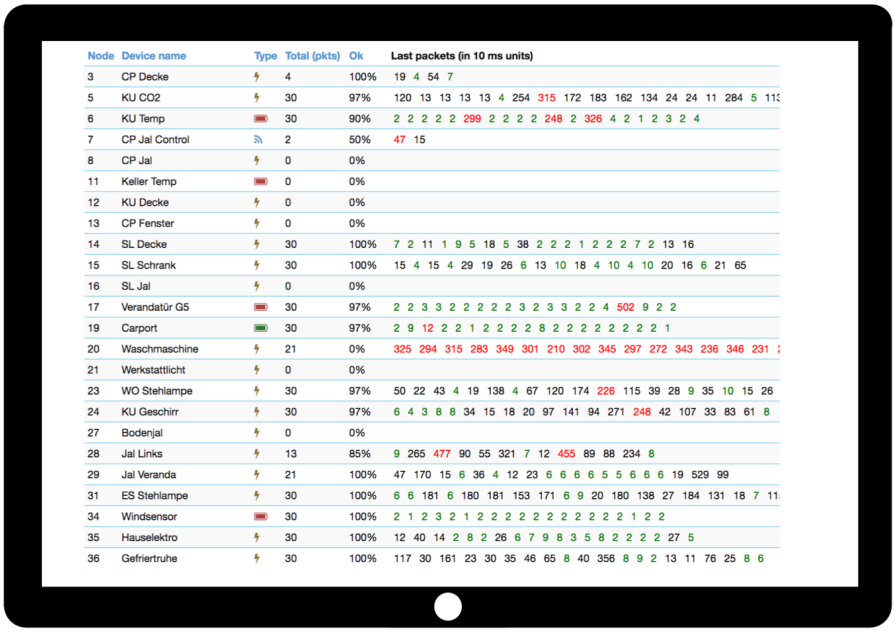
- Node 5 can be reached via routes only but one time not even this worked. There was some error. It is possible that the failure of node 20 caused his and then the system found an alternative route.
- Node 6 seems to be in direct range with very stable communication but from time to time there is a failed communication. Since this is a battery-operated node, it is highly likely that the last communication with the device reaches this device while already in deep sleep state. This does not harm the communication at all but is worth monitoring.
- Node 15 switches between direct communication and routed communication. It seems to be right on the edge of having a stable direct link but sometimes - may be when doors are open/closed - the direct range does not work anymore.
  Anyway, the controller seems to understand that direct range is the by far best option and constantly tries to reach the node in direct range. The same pattern can be seen for nodes 29 and 31.
- Node 24 has an interesting history. For some time, there was a stable direct range communication but then it got worse and worse to a point where communication even failed. However, the link recovered and the very last communication was again direct, but with a slight delay of 80 ms.

8.3 Network Layer - Weak or Wrong Routes

It is the best already knowing the troublemaking devices. In this case the status of device can be checked quickly and it is possible to dig deeper into the routing layer. Figure 8.9 shows the routing table of a controller. Technically this is not a routing table but a matrix indicating the wireless neighborhoods of devices. Nevertheless, this a good starting point to investigate deeper. Having many neighbors is a good thing since the routing algorithm has many options in case something goes wrong. On the other hand, just having one other route to communicate to the rest of the network may cause trouble if this route is faulty or moved.

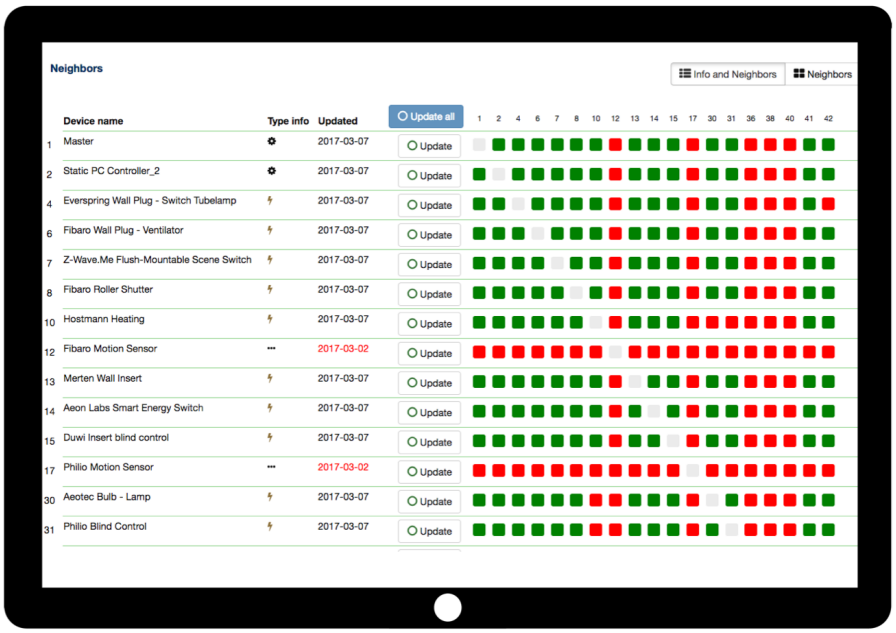
The next step is to check individual routes. The configuration page of every device offers a link health check that allows testing the links from this very device to its neighbors.

While the neighborhood table shows if two devices are neighbors, the link test checks how good this wireless links is. Unfortunately, not all but an increasing number of devices on the market support this link test. Figure 8.10 shows



Node	Device name	Type	Total (pkts)	Ok	Last packets (in 10 ms units)
3	CP Decke		4	100%	19 4 54 7
5	KU CO2		30	97%	120 13 13 13 13 4 254 315 172 183 162 134 24 24 11 284 5 11
6	KU Temp		30	90%	2 2 2 2 2 299 2 2 2 2 248 2 326 4 2 1 2 3 2 4
7	CP Jal Control		2	50%	47 15
8	CP Jal		0	0%	
11	Keller Temp		0	0%	
12	KU Decke		0	0%	
13	CP Fenster		0	0%	
14	SL Decke		30	100%	7 2 11 1 9 5 18 5 38 2 2 2 1 2 2 2 7 2 13 16
15	SL Schrank		30	100%	15 4 15 4 29 19 26 6 13 10 18 4 10 4 10 20 16 6 21 65
16	SL Jal		0	0%	
17	Verandatür G5		30	97%	2 2 3 3 2 2 2 2 2 3 2 3 3 2 2 4 502 9 2 2
19	Carport		30	97%	2 9 12 2 2 1 2 2 2 2 8 2 2 2 2 2 2 2 2 1
20	Waschmaschine		21	0%	325 294 315 283 349 301 210 302 345 297 272 343 236 346 231
21	Werkstattlicht		0	0%	
23	WO Stehlampe		30	97%	50 22 43 4 19 138 4 67 120 174 226 115 39 28 9 35 10 15 26
24	KU Geschirr		30	97%	6 4 3 8 8 34 15 18 20 97 141 94 271 248 42 107 33 83 61 8
27	Bodenjal		0	0%	
28	Jal Links		13	85%	9 265 477 90 55 321 7 12 455 89 88 234 8
29	Jal Veranda		21	100%	47 170 15 6 36 4 12 23 6 6 6 6 5 5 6 6 6 19 529 99
31	ES Stehlampe		30	100%	6 6 181 6 180 181 153 171 6 9 20 180 138 27 184 131 18 7 11
34	Windsensor		30	100%	2 1 2 3 2 1 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2
35	Hauselektro		30	100%	12 40 14 2 8 2 26 6 7 9 8 3 5 8 2 2 2 2 27 5
36	Gefriertruhe		30	100%	117 30 161 23 30 35 46 65 8 40 356 8 9 2 13 11 76 25 8 6

Figure 8.8: Paket timing of an aged Z-Wave network



Neighbors

Info and Neighbors | Neighbors

	Device name	Type info	Updated	Update all	1 2 4 6 7 8 10 12 13 14 15 17 30 31 36 38 40 41 42
1	Master		2017-03-07	Update	
2	Static PC Controller_2		2017-03-07	Update	
4	Everspring Wall Plug - Switch Tubelamp		2017-03-07	Update	
6	Fibaro Wall Plug - Ventilator		2017-03-07	Update	
7	Z-Wave.Me Flush-Mountable Scene Switch		2017-03-07	Update	
8	Fibaro Roller Shutter		2017-03-07	Update	
10	Hostmann Heating		2017-03-07	Update	
12	Fibaro Motion Sensor		2017-03-02	Update	
13	Merten Wall Insert		2017-03-07	Update	
14	Aeon Labs Smart Energy Switch		2017-03-07	Update	
15	Duwi Insert blind control		2017-03-07	Update	
17	Philio Motion Sensor		2017-03-02	Update	
30	Aeotec Bulb - Lamp		2017-03-07	Update	
31	Philio Blind Control		2017-03-07	Update	

Figure 8.9: Neighbor-Table of a controller

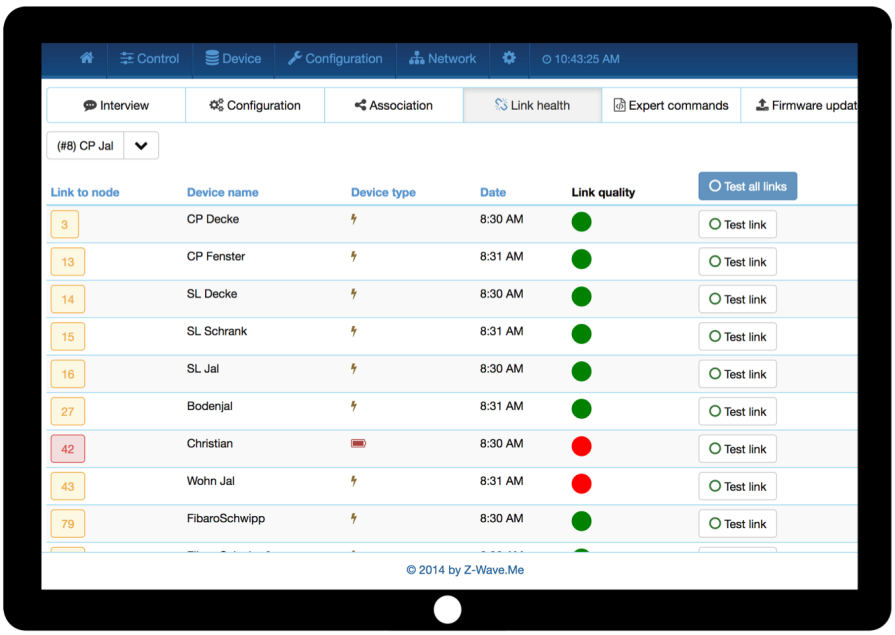


Figure 8.10: Link test of a node

this dialog within **Z-Wave Expert User Interface** . Every link has a color indicator (green = ok, red = bad, grey = unknown) and a time stamp that shows when this test was done the last time.
Please note that the link check is a momentary analysis only and does not give any information about the history of the link quality.

8.4 Application Layer Settings

In the application layer, there is usually no malfunction of a device but wrong configurations. **Z-Wave Expert User Interface** allows changing and monitoring the values.

8.4.1 Polling

Heavy polling of devices causes network traffic leading to delays. A simple look on the sniffer as shown in Figure 8.6 reveal if there is too much polling.

8.4.2 Dead Associations

Association enable direct communication between devices. In case there are more than one device in an association group, they will receive a command one after each other. A very common problem is that associations are set during the built-up of the network and later certain devices are removed or simply fail. If this disappeared node is still in an association group, the device will always try to communicate to this node first before communicating to other nodes. The result is a delay. The device-specific configuration overview as shown in Figure 8.11 displays all association that are set. It is possible to recall the current associations from the device and to remove or set associations.

8.4.3 Wrong Wakeup Settings

Wrong wakeup settings may either result in too much traffic draining the battery, or in too slow response to sensor update requests or configuration changes. The status overview page as shown in Figure 8.5 gives a simple overview of the wakeup settings of the different battery-operated sleeping devices. The device-specific configuration settings allow changing these settings. Besides the wakeup interval, the setting also allows setting/changing the Node ID of the controller holding the mailbox of this device. This setting must reflect the correct situation in the network.

8.5 Summary

Table 8.1 summarizes the possible *"10 root causes of Z-Wave network problems"* and suggestions how to fix them.

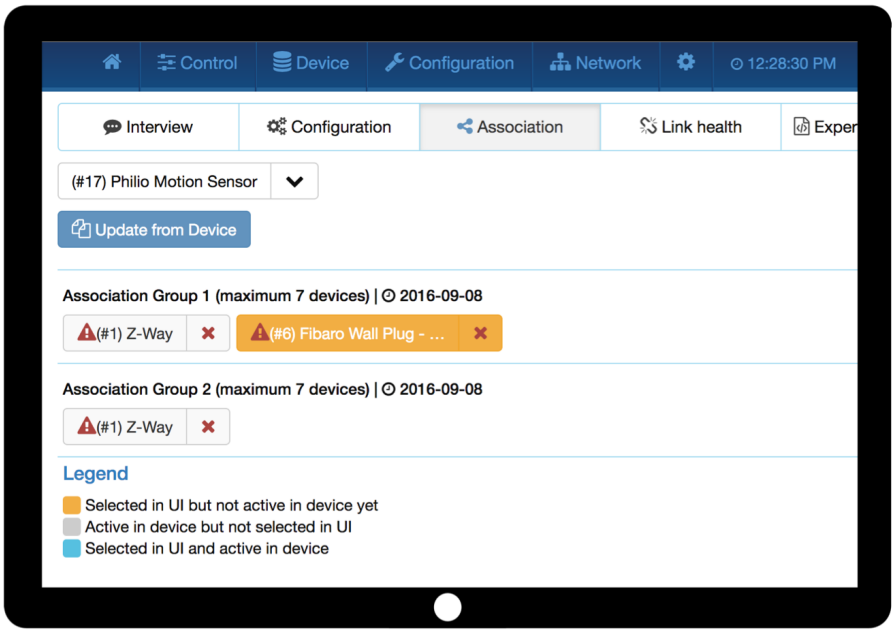


Figure 8.11: Association Dialog in **Z-Wave Expert User Interface**

No.	Cause	How to find ?	How to fix ?
1	Noise by other transmitters	Background Noise Chart	Find them and turn them off
2	Noise by other Z-Wave networks	Background Noise Chart, Network Statistics	Talk to the neighbor ;-)
3	Faulty devices	Status Page, Failed Node	Remove them or replace them.
4	Crazy Devices (always sending)	Sniffer	Remove them or replace them
5	Weak Link	Neighbor-Table, Link Health in Configuration Page	Add more routing nodes, move devices
6	Heavy Fading	Timing Infos	Network Reorganization, more devices
7	Wrong Routing	Timing Infos	Network Reorganization
8	Wrong Polling	Sniffer	Change and Save
9	Wrong Wakeup Intervals	Status Page	Change and Save
10	Dead Nodes in Assoc. Groups	Association display in Configuration Page	Change and Save

Table 8.1: Troubleshooting on Z-Wave networks

9 Extending the systems beyond Z-Wave

9.1 IP-Cameras

IP cameras transmit a video stream that can usually be accessed having dedicated mobile apps. Under certain circumstances it is possible to have the very same video stream in parallel within the Z-Way Smart Home Interface . All Z-Way user interfaces (Web Browser and native apps for IOS and Android) are based on off-the-shelf HTML rendering engines supporting standard video and image formats. To display the video stream of a certain camera this stream must comply to commonly used public standards, such as **MJPEG**.
Certain cameras however use **proprietary video encoding** that can only be decoded by special native mobile apps of the manufacturers. These cameras can't be supported by Z-Way.

9.1.1 How to find out if a camera is supported by Z-Way?

1. Check the manual if MJPEG is mentioned as encoding method for the video stream.
2. Check if there is a way to access the video stream using a standard web browser such as Google Chrome or Microsoft Internet Explorer.

9.1.2 How to prepare for integration?

To integrate a camera into Z-Way, this camera needs to be setup first following the guidelines given in the manufacturers manual. As a result, there must be

- A login name (examples are "admin" or "user")
- A password for access (this usually needs to be setup
- The IP address of the camera

9.1.3 How to find the IP address of the camera?

Most IP networks in private homes and offices assign IP devices a new IP address using the DHCP protocol. The router holds a list of IP addresses and will arbitrarily choose one address from the list to the new IP device. Even after a reboot this address will remain the same. If the setup process mentioned above does not reveal the IP address there are two common ways:

- Log into your IP routers user interface. Usually this interface will display all IP addresses assigned together with a name and/or type of the device.
- Use an IP address scanner on your PC or notebook that will tell you all IP devices active in a network plus the type of IP device they ae assigned to. A valuable tool for this is called "Angry IP Scanner" available for all PC platforms such as MAC OSX, Linux or Windows. It requires a Java Virtual Machine (JVM).

9.1.4 How to integrate the camera into Z-Way?

The Z-Way Smart Home Interface allows adding new devices. Log into the user interface of Z-Way, click on the setup menu (icon on the upper right side) and click on menu item Add New Device . You see the dialog as shown in Figure 9.1. Now choose the IP camera symbol line and click + .
Now you will find a list of IP camera types plus one generic IP camera option called **'Web Camera'**. This dialog is shown in Figure 9.2.
If you are lucky, the name of your camera is already on the list. If not, you can check in the app store if there is a new support app available for your camera type. Go to Setup » App » Online Apps and filter for **'Video Surveillance.'** Figure 9.3 shows the app store with camera filter.
If you find your camera type, just install the app and redo the steps above. Now you will find the new camera in the list to choose from.
Once you click on the camera of your choice there will be a setup wizard asking you as a minimum for the IP address of your camera, the login name, and the password. Therefore, you need this set of data from the setup process of your device.

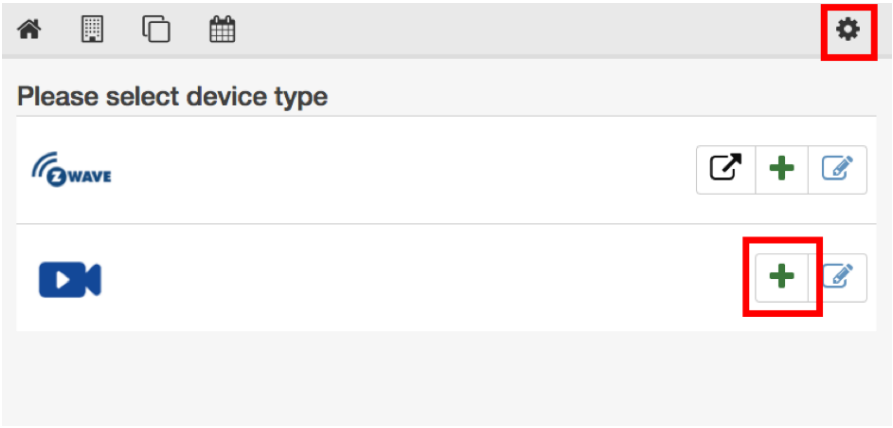


Figure 9.1: Inclusion of predefined cameras

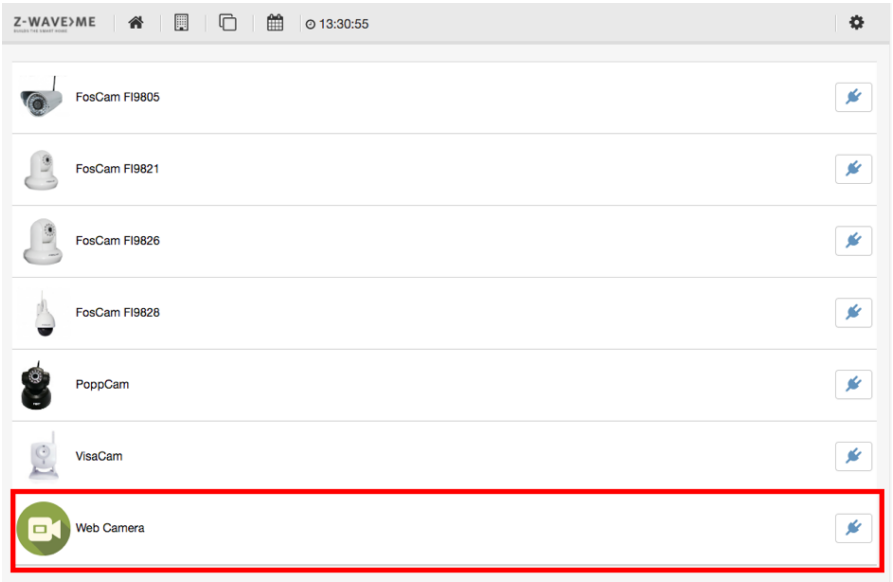


Figure 9.2: Generic camera module

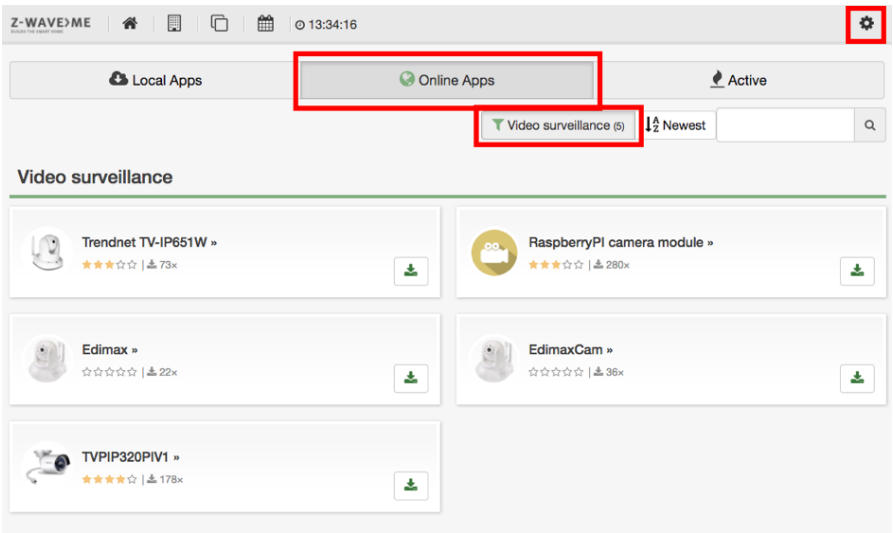


Figure 9.3: More camera support in App Store

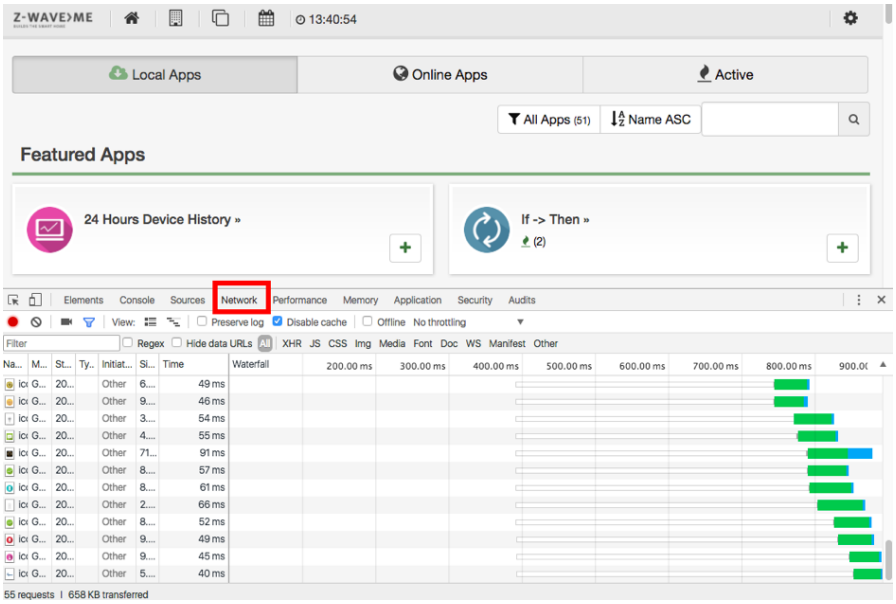


Figure 9.4: Web browser debug interface

9.1.5 How to support a camera not on the list yet?

If your camera type is neither on the list of preset cameras nor there is a new app in the online app store there is still a very good chance to get your camera integrated. However now there is more work needed to find the right commands controlling your camera. In this case, you will need to choose the generic camera type "Web camera," which requires the same set of information (IP address, login, password) but more than that.

Note: This work requires some basic understanding of web pages, IP, and URLs. The generic web camera allows defining the URL to the video stream and—if the camera has these capabilities—links to tilt, turn, night vision control, etc.

To find these URLs, you need to log into your camera using a generic web browser. We strongly recommend using Google Chrome because of the debugging capabilities. The following explanation assumes the Chrome browser, but other browser will have similar functions.

1. Open the JavaScript Debug console. You find this option on the browsers menu under View > Developer. Figure 9.4 shows the web browser with debugger active.
2. Once the debugger is open pick the menu item network of the debugger (see image below"
3. Now you use the cameras web interface for accessing the image/stream, tilting, moving, etc. Whenever you do this the URL needed will be sent from the web interface to the camera and becomes visible in the debugger. Take these URLs and copy them into the setup interface of **'Web Camera'**. The camera control in Z-Wave will call the same URL for control.

9.2 433 MHz devices

9.2.1 Introduction

433 MHz wireless communication is an outdated wireless technology. It is single direction wireless communication only without confirmation of received packets and it does not have any security functions. It is not standardized and subject to jamming by other devices since 433 MHz is a non-regulated frequency range. However, it is still used is many low-end alarm and control systems and there is a quite large install base in the market. The biggest advantage of 433 MHz devices is its low price.

Due to its one-way wireless connection, sensors can only report values and actuators can only receive values. Unfortunately, there is no regulation in the frequency band. Every supplier has its own code set for actuators and sensors. Typical suppliers of 433 MHz devices are Intertechno, Conecto, Mumbi, Homeeasy, Elro, Teldus, Brennenstuhl, Olympia and others.

9.2.2 433 MHz Gateway

In order to support 433 MHz devices special gateways are required. Z-Way has built-in support for the 433 MHz gateway from Popp. Figure 9.5 shows this gateway hardware. The device is powered by an external standardized mini

Figure 9.5: Popp 433 MHz Gateway

USB port and can be connected ot USB wall outlets, mini USB power supplies, etc.
Once configured and connected to the Z-Way System, this gateway allows to learn different code sets of different manufacturers. It can therefore be used universally for all kinds of 433 MHz devices from different manufacturers even if there is no detailed technical description of the code set used.
Z-Way can support multiple Popp 433 MHz gateways when one single gateway cannot cover the whole home. The Popp 433 MHz hub is connected to Z-Way using Wi-Fi. This means that there must be at least one Wi-Fi network to connect the 433 MHz and an IP connection to the Z-Way system (not necessarily Wi-Fi but cable Ethernet is possible too if there is a router between the cabled ethernet and the Wi-Fi).

9.2.3 How to setup the 433 MHz Gateway

First, install the app **'RF433'** from the online app store. For more information about the online app store, please refer to Chapter 6.
Activate the RF433 app as shown in Figure 9.6. If you like you can change the operating IP from 8000 to any other IP port number available. However, its perfectly fine to keep it at 8000. Once the RF433 app is running on your Z-Way system you need to setup the 433 MHz gateway.

Power Up the 433 MHz Gateway

Place the 433 MHz connector on the place of choice and power it using a standard 5V USB power supply. Push the central button until the LED slowly blinks in purple indicating that the device is in configuration mode serving its own access point.
In this mode, the gateway acts as access point creating its own Wi-Fi network. The SSID of this Wi-Fi network is **gw433-xxx** with xx as some individual serial code.
You need to connect to this Wi-Fi network using any Wi-Fi capable device available (e.g. mobile phone, notebook, etc.). Now start a web browser on this device and open the page

`http://192.168.4.1`

to access the configuration interface as shown in Figure 9.7. There is no password needed.

Configure the 433 MHz Gateway

Click on Configuration to access the setup dialog as shown in figure 9.8.

- **Wifi-Settings**: Choose the SSID of your Homes Wi-Fi. This is the Wi-Fi that establishes the IP connection between the 433 MHz Gateway and the Z-Way controller. You can scan for available SSIDs. In case the Wi-Fi network selected requires a password (or WPA key) enter this key as well.
- **DHCP Active**: In most case the Wi-Fi you connect to will run DHCP for automated IP address assignment. Only if you know for sure that there is no DHCP you need to setup a fixed IP address plus network mark plus gateway.

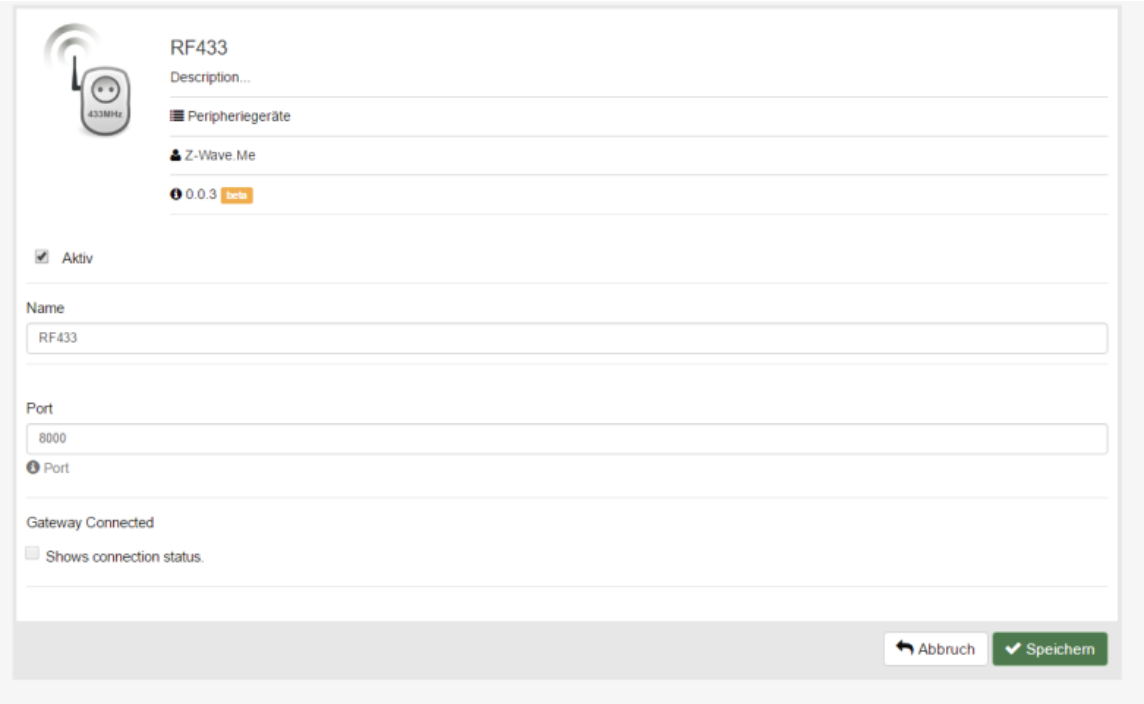


Figure 9.6: RF433 App Setup

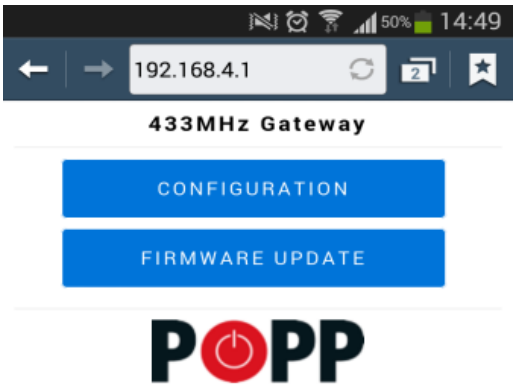


Figure 9.7: 433 MHz gateway web interface

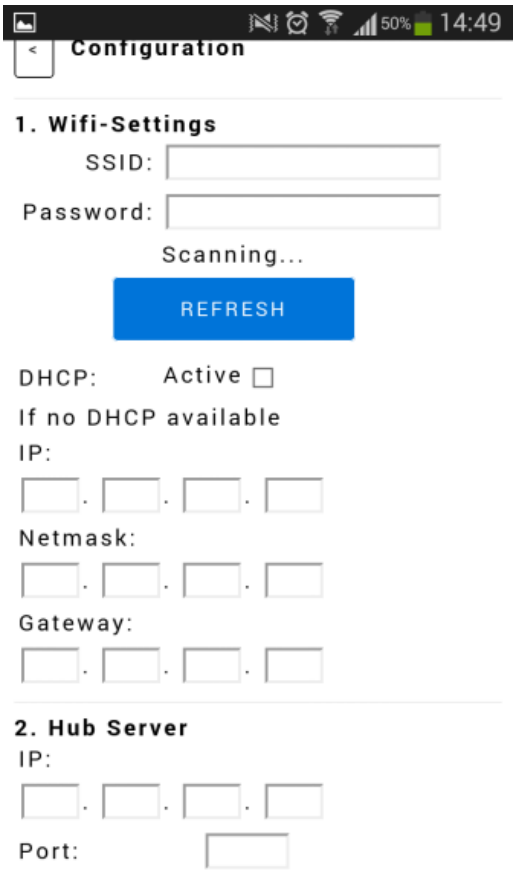


Figure 9.8: 433 MHz gateway setup dialog

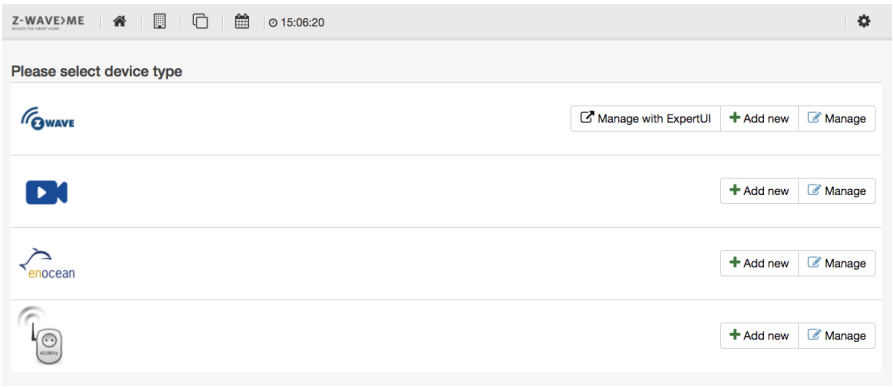


Figure 9.9: 433 MHz option in 'Devices'

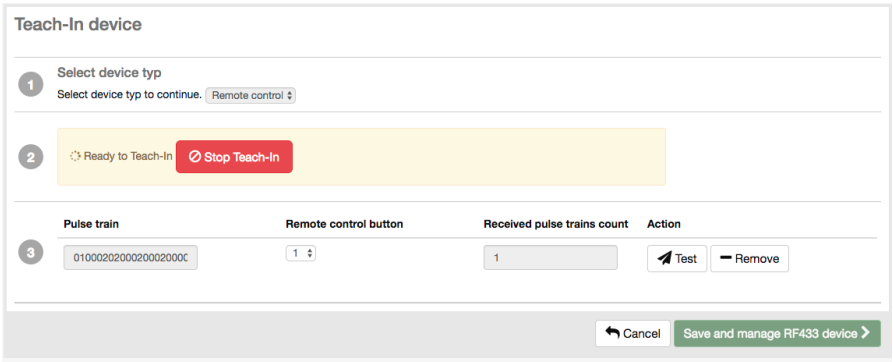


Figure 9.10: 433 MHz teach in

- **Fixed IP Settings**: This setup is only needed if there is no DHCP service on the Wi-Fi selected.
- **Hub Server/Port**: Here you need to configure the IP address of your Z-Way controller and the IP port address chosen during the setup of the RF433 app on Z-Way.

Activate the connection mode of the Gateway. The local Access point will be deactivated and the gateway connects (1) to the Wi-Fi and through the Wi-Fi to the (2) Z-Way controller. Success is indicated by blinking of the green LED. You can always return to the configuration mode by a long push of the central button on the 433 MHz gateway. In this case any connection to the Z-Way controller is deactivated and the local Wi-Fi with SSID **gw433-xxx** is active again. Here again the different LED codes of the Popp 433 MHz gateway:

- blue blinking: Configured but searching for connection to Z-Way controller
- purple blinking: Gateway is in initial configuration mode. Connect to Wi-Fi APN gw433-XXXX and call URL http://192.168.4.1 for initial configuration
- green permanent: connection established
- green short blink: communication from/to gateway

Teach In 433 MHz devices

Once the 433 MHz gateway is configured corrected and is connected it is possible to teach-in 433 MHz devices. A new section of devices will appear on the "Device" overview in the setup menu of the user interface as shown in Figure 9.9. After clicking the Add button choose the type of 433 MHz device to teach in:

1. Binary Sensors such as door sensors and motion detectors
2. Remote Controls
3. Actuators like Smart Plugs

While Sensors and Remote controls only send out commands the actuators receive commands only. To teach them into the system a little trick is needed. Each Smart Plug or other actuator comes with a small remote control sending exactly the commands expected by the actuator. In order to control an actuator, the associated remote control is required to issue commands that can be captured by Z-Way.

The remote control itself is handled in the same way. The only difference is the way the elements are created.

Once device type is selected and the teach in process has started, all the buttons of the remote control must be pressed,

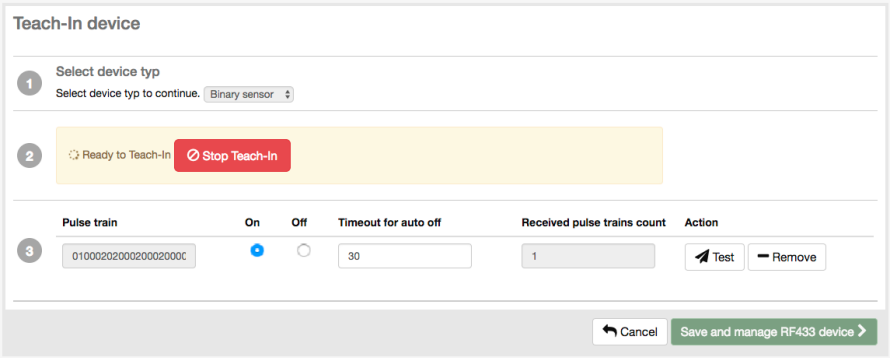


Figure 9.11: 433 MHz teach in of a binary sensor

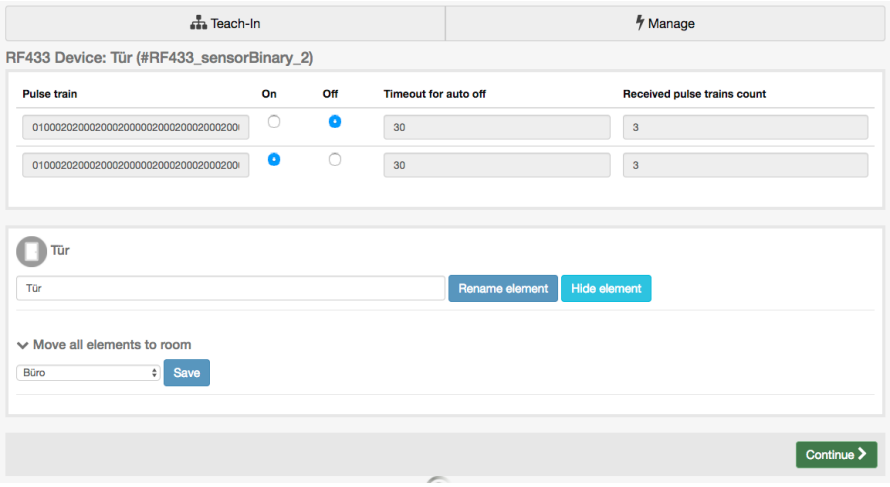


Figure 9.12: 433 MHz device management

one after each other. Figure 9.10 shows this moment. The pulse train of the button commands are shown in the tech-in dialog and the number of the remote-control buttons can be assigned.
For Actuators the same process applied, but the status of the actuator needs to be assigned to the pulse train.
Another specialty concerns binary sensors. Some 433 MHz sensors send signals on open and on close. However, some other sensors only create one wireless command when the sensor trips. For alarm systems, this is enough but for Smart Home with User Interfaces this is not working since the element cannot show the actual status. For these kinds of sensors, there is an automatic switch back to off function after a defined time interval. Figure 9.11
Once all pulse trains are captured, the new element need to be renames and assigned into a room.
The management function for 433 MHz devices allows accessing the setup and change it as shown in figure 9.12

9.3 EnOcean devices

EnOcean is another wireless communication technology optimized for very low power consumption.
Z-Way has implemented support for EnOcean devices but limits its function to sensors and wall switches because of their battery-free and therefore maintenance-free design. Compared to Z-Wave, EnOcean is a quite simple protocol. There is no such thing like network inclusion or routing—every EnOcean device just sends out a specific datagram that includes a unique device id and the data (sensor values, switch status) of the specific function of the device. The encoding of these data is defined in so-called profiles. These profiles are identified by a three-byte value but they are not transmitted wirelessly. Hence the user must decide from his product knowledge what profile a certain device is using. The EnOcean receiver will use this information to decode the datagram and use the data. (This means that a wrong decision about the profile of a EnOcean device will lead to severe malfunctions of the system).
Every EnOcean receiver in proximity will always receive every datagram sent by a transmitter. This leads to two basic management functions of the EnOcean module:

1. select the right products (by their unique 4 Byte ID) to use—and ignore all others

Figure 9.13: Popp EnOcean USB Stick

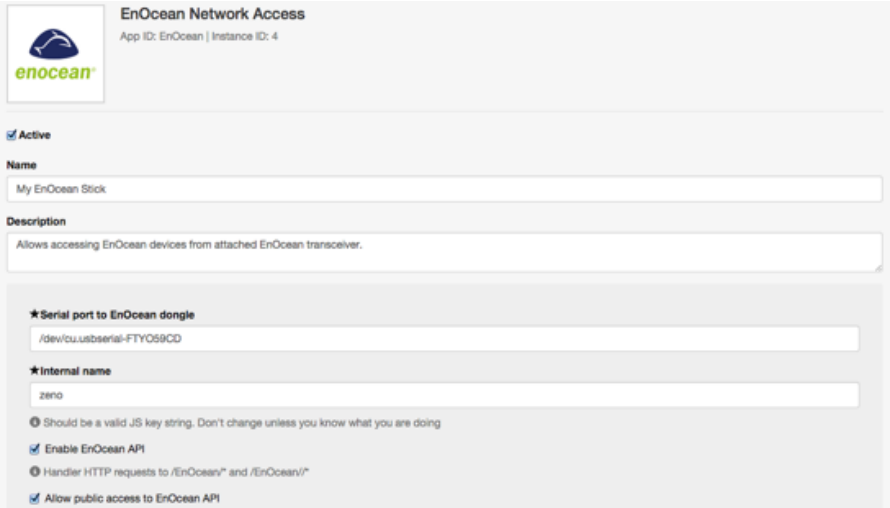


Figure 9.14: EnOcean App configuration

2. define the correct profile by selecting the right product

To work with EnOcean devices, an EnOcean USB Stick is required. Please use the Popp EnOcean Stick (POPE12204) as shown in Figure 9.13 and plug it into the USB port[1].

Next, the EnOcean app must be installed from the app store and configured as shown in Figure 9.14.

Make sure to pick the right device name of the EnOcean USB Stick connected to your hardware. For Raspberry Pi-based platforms this is always `/dev/USB0` but for other platforms this may be different. The internal name, "zeno," can be arbitrarily chosen. In case more than one EnOcean stick is operated this name needs to be unique.

Now it is possible to "teach-in" new products using the user interfaces "Device" section Configuration » Device » EnOcean. As described above the first step is to select the right product. A list of manufacturers with their products are given to select from. Please note that the EnOcean module may support many more devices from other manufacturers as long as they have the same profile. A good example for this is a door window sensor (profile name D5-00-01). Multiple manufacturers off door-window sensors, some even in the same enclosure but some in slightly changed enclosure. Their EnOcean wireless capability is however similar.

You may find the profile name on the label of the unknown device or documented in the device specification of the manufacturer.

Please not that without proper knowledge of the device and its profile it is impossible to operate the device!

After selecting the right device, the user interface asks for the teach-in process as shown in Figure 9.15. During this process, the new device must send out one datagram containing the unique ID. The user interface will give some hints how to generate such a datagram.

Once this datagram was received, the EnOcean module will generate virtual devices according to the profile selected. You can change the names of the elements to be generated. Figure 9.16 shows this dialog.

Finally, as shown in Figure 9.17 one or multiple elements will appear in the elements view. One example of the wall controller device shows Figure 9.18.

[1] Other EnOcean sticks may work as well, but the correct function is not supported and they may stop working after Z-Way firmware updates.

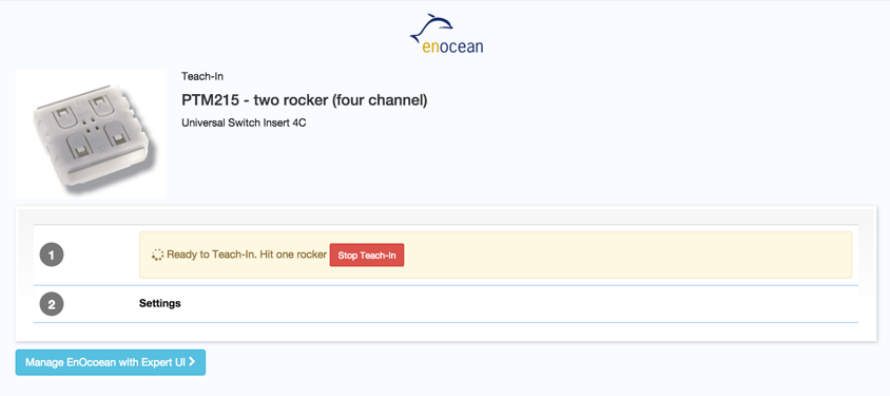


Figure 9.15: EnOcean Teach In

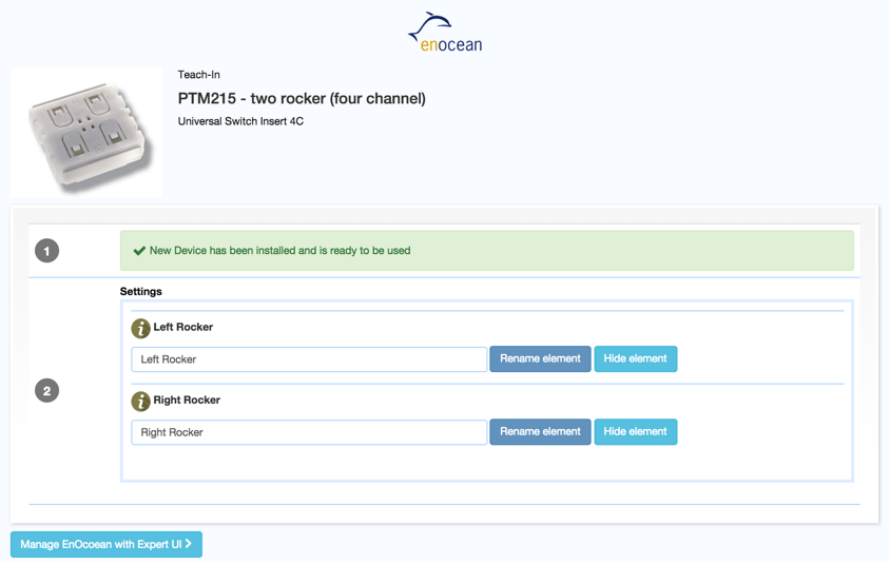


Figure 9.16: EnOcean Device Configuration after Teach-In

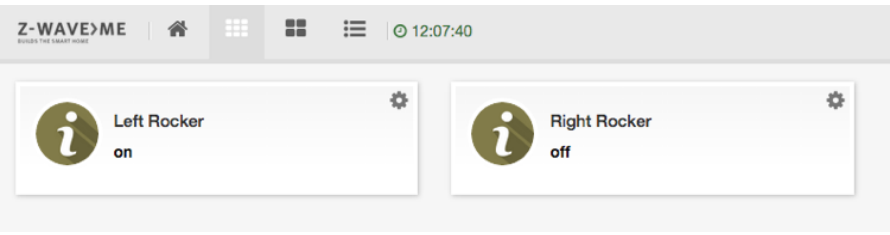


Figure 9.17: EnOcean Device Elements

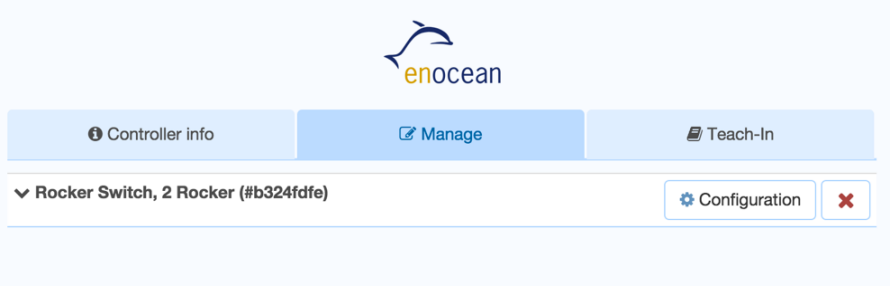


Figure 9.18: EnOcean Device Management

The menu option Devices also offers a special interface to manage EnOcean devices as shown in Figure 9.18. This dialog offers a list of all known EnOcean devices with an option to change the profile. Furthermore, it is possible to manually select one of the valid profiles of EnOcean for a device not known to the standard user interface. Please note that profiles not known to the standard UI are also not supported by the standard user interface and will not leads to creating new UI elements. However, the device is still created in the API and can be used by third-party software.
For a list of all supported EnOcean devices, please refer to Annex D. Experienced Users and programmers may extend this list by adding their own profiles to Z-Way. Chapter 13.4 describes how to do this.

9.4 Other IP/Internet-based services

Z-Way can work with external IP-based systems. Please refer to the app store description for more information about how to integrate third-party IP-based devices. Please refer to Chapter 6 for details.

10 Customize your system

10.1 Skins

Wouldn't it be cool to have your own individual user interface controlling your own individually designed Smart Home? Z-Way offers you exactly this feature—it is called **'Skin'**. The Skin is a software package redefining all visual elements of your mobile and browser interface including images, fonts, colors, wallpaper, etc. Figure 10.1 shows the menu option for customization on the setup menu in the user interface.
Designing a new skin from scratch is a lot of work. It is easier to choose an already existing skin. Go to Setup › Management › Customize and activate a new skin. You can also download skins from the online server.

10.1.1 Step 1 - Do you own Skin

A skin consists of a set of images and a description file for fonts, colors, etc., called CSS (Cascaded Style Sheets). See Annex A for links to more information about CSS.
The starting point for a new skin is a blank template you can download from

`http://github/z-wave-me/Skin-blank`

.
This file is a zip archive you need to unzip into a temporary folder. This folder contains two sub folders:

- **/blank**:
  This is the blank skin template including the CSS file main.css, a screenshot image for the selection in the store and a subdirectory with all the images needed for the skin.
- **/sass**:
  This is the source code to generate the main.css—more on this magic later!

10.1.2 Step 2 - Do your own Images

Images are a central part of any skin. As shown in Figure 10.2 the /blank/img subdirectory contains two sub-directories and two files:

- **/icons**:
  The images of all the different elements. Please be aware that some element types like dimmers have three, some have two, and some have only one icon. The names of the icons are self-explaining.
- **/logo**:
  This contains the logo displayed on the upper left side of the screen and the wallpaper.
- main.css: This is the cascaded style sheet you will need to edit.

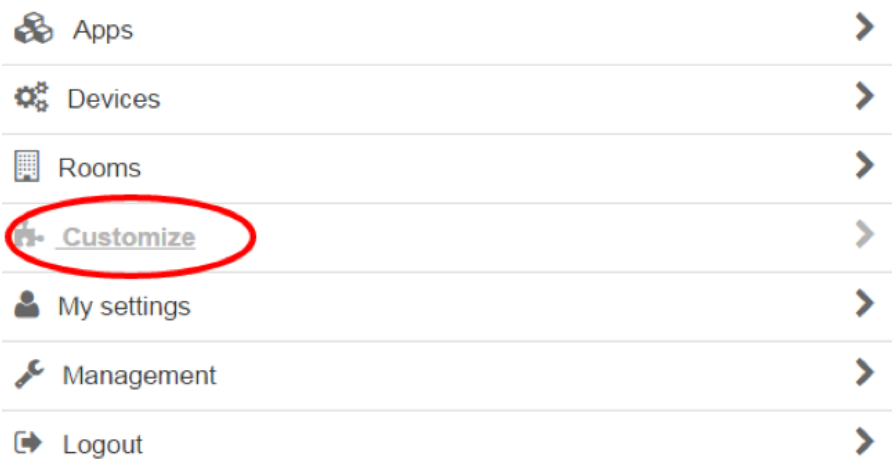


Figure 10.1: Skin Setup

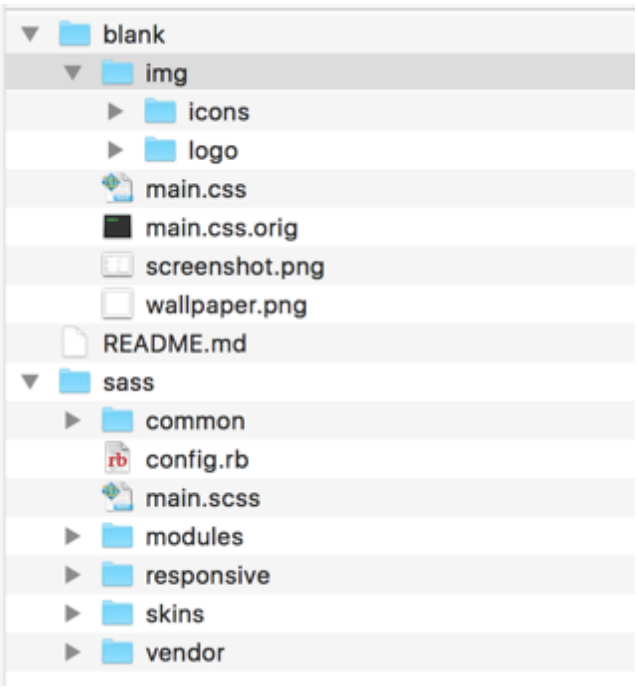


Figure 10.2: Skin directory structue

- main.css.orig: This is your safety belt. In case you mess up your main.css here you have the original as backup.
- Screenshot.png: The preview image for selecting a skin
- Wallpaper.png: the wallpaper of the User Interface

There are plenty of tools to redesign and change images. This short write up will not explain this in detail but the Internet if full of resources for image editing. Just a few remarks:

- Use exactly the names of the icons as they are provided in the blank skins. Otherwise, they will not be used.
- Icons should be 64x64 pixels. Make them as small as possible allow fast loading

Hint: The easiest change of a skin is just to replace the wallpaper image by something individual.

10.1.3 Step 3 - Test the new Skin

A quick way to test a new skin is to load it directly to your Smart Home gateway running the Z-Way controller software. This is possible for all Z-Way installations running on a PC (Linux, Windows) or on a Raspberry PI. Z-Way installations on other "closed" boxes such as Popp Hub are not suited for such a quick test drive—sorry!

First of all you **need to choose the "Default Skin"** in your Z-Way Smart Home Interface on Setup › Management › Customize. **Only the Default Skin** can be changed in the quick way described below.

Then you need a way to copy files of your new Skin on the Z-Way installation. This can be done using simple file copy (when developing on the PC running Z-Way) or using FTP. You can replace all images by copying them into the folder.

```
/opt/z-way-server/automation/storage/images/
```

.

One exception is the wallpager.png which is in the folder

```
/opt/z-way-server/htdocs/smarthome/app/css
```

like the main.css that holds all other settings.

Once a new file is changed or uploaded reload the UI on your browser or restart your native app, and voila, your changes are visible.

Hint: To test a new wallpaper, just copy your file of choice to

```
/opt/z-way-server/htdocs/smarthome/app/css/wallpaper.png
```

and reload the page.

10.1.4 Step 4 - Change colors, fonts, shapes – almost

Colors, fonts, etc. are all controlled by the file main.css. Open this file in a text editor and you will be shocked by the about 10.000 lines of code. If you are a CSS pro you may be able to edit this but this is not the recommended way to do this. CSS is great tool for shaping web pages but it has much legacy that makes it hard to edit manuals. However, if you really like to go that route Annex B will provide you some hints where to find the important lines of code. Use at your own risk—you are warned.

10.1.5 Step 5 - Going into the SASS world

SASS is a preprocessor for generating CSS files. It extends CSS syntax and adds a few very useful functions such as central variables. This and a few other advantages caused many web designers moving away from writing CSS directly but using SASS.
The disadvantage is that you need to have another software on your developer PC translating the SASS files into the final CSS (main.css). Annex B provides some links to SASS tutorials and to some tools to translate from SASS into CSS.
We recommend the tool "Scout" because it works equally well on Windows and on MAC, is well documented and does most of the magic automatically.

- Download Scout from http://scout-app.io/ and install the tool
- Watch the movie

  `www.youtube.com/watch?v=Fju3aXW6zLM&feature=youtu.be`

  for instructions on how to set up and use Scout.

Few Hints:

- Start Scout and setup as shown in the movie. Point to the folder /sass as source folder and to the final folder of your skin as destination.
- The generated main.css file you can upload to your test box as described above. Whenever you change a SASS file the Scout application will detect it and automatically update the generated main.css. Then you can upload the main.css to your test controller.

10.1.6 Step 6 - Changing SASS

Finally we come to the point of making a real new Skin by changing the layout, color and font definitions of the blank skin. For this you need to edit the sass files provided with the blank skin. The central sass file is main.scss. It does not contain any layout definitions but loads all the other needed files only. The idea behind sass is among others to have different functions separated into different files. For the Skin in Z-Way however 98 % of all changes will happen in only one file - /common/_variables.scss. This file contains all major definitions for colors, shapes, sizes, fonts, etc.
Hint: A first run should be like:

1. Change an important color, e.g. $app-color-primary: #000000;
2. Save the file
3. Watch Scout compiling the change and updating main.css
4. Uploading the new main.css to the test box
5. Reload (and empty cache !!!) the web page and see the result

10.1.7 Step 7 - Create the final Skin for friends, family and the public

In order to create a final skin file, a few more work needs to be done.

Create a preview image

Just make a screenshot of the new skin as preview. For the skin selection in the Z-Way Smart Home Interface the screenshot must be stored in the folder skinname/ and must be named screenshot.png Recommended dimension for this image is 300px X 150px.

Collect all files together

You need a folder with the name of the skin. This contains the screenshot.png, the wallpager.png and the main.css plus the subfolder (img that has two further subfolders /logos and /icons)

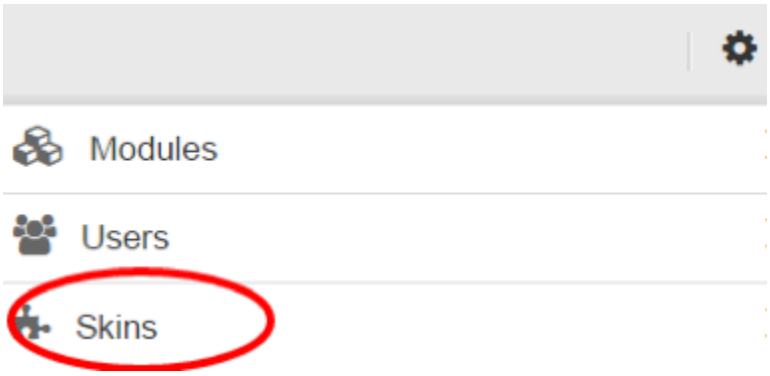


Figure 10.3: Go to menu Skin

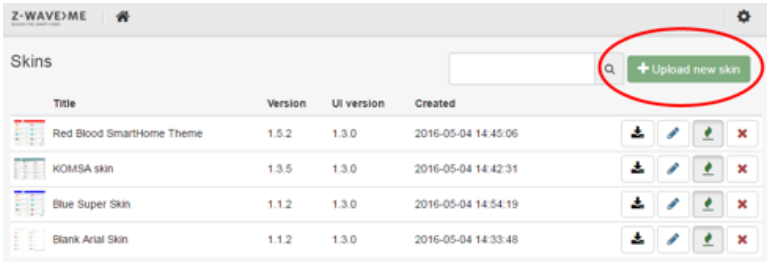


Figure 10.4: Upload new Skin

```
myskin
├─ main.css
├─ myskin.png
├─ wallpaper.png
└─ img
   ├─ logo/*.png
   └─ icons/*.png
```

Pack the files

The whole folder needs to be packed now using ZIP archive program. On MAC please make sure to use the ZIP command line and not the built-in compression tool of the finder. This will not work. Move inside the folder of your skin (example above is /myskin) and execute “zip -r -X myskin.zip *.”

10.1.8 Step 8 - Distribute your Skin

1. If not done yet create your personal account on the https://developer.z-wave.me/
2. Go to Menu -> Skins as shown in Figure 10.3.
3. Click on the “Upload new skin” button as shown in Figure 10.4.
4. Select a packed skin from your PC as shown in Figure 10.5.
5. Skin will be automatically uploaded. If an upload process is successful an update form is shown. In the form activate skin, enter title, UI version, Skin version, description, author name, homepage and upload a skin image. Click on update.

10.1.9 Step 9 - Rewind in case something goes wrong

For sure you will end up with skin attempts that don’t work well. In a worst-case scenario, you can’t even pick a different skin anymore and your default Skin was messed up. For such a case, there is an emergency reset Just call

Figure 10.5: Select the packed Skin

the URL

```
http://IP:8083/ZAutomation/api/v1/skins/setToDefault
```

.

10.2 Icon Sets

It is possible to add individual icon sets to the Z-Way and share it with others.

10.2.1 Create Your own Icons

First to the icon as such. They must have a size of 64x64 pixel and must be encoded in PNG image file format. You are free to make every icon you can imagine. Please note that there is a list of typical devices where standard icons are applied:

- battery
- heating
- motion
- energy
- water
- gas
- switch
- smoke
- door
- window
- light
- media
- blinds
- cooling
- co
- fan
- flood
- thermostat
- luminosity
- humidity
- temperature

Please note as well that certain elements need two or even three icons to indicate different status of their operation.

10.2.2 Create an Icon Pack

It is certainly possible to just replace the icons right in the User interface exchanging the file in the filesystem on /opt/zway-server but there is a much better way. Icons should be grouped into icon sets and should be placed on the server to be available for all Z-Way users. The grouped icons are called icon packs.

An icon pack is essentially a gz or zip archive file containing different icons. All icons need to be stored in a subfolder with the name of the icon set. The archive must then have the very same name. Only store the icons in the archive and not the subfolder itself! or Unix OS these commands will work:

1. cd /icons/youriconpack
2. tar -cvzf youriconpack.gz *

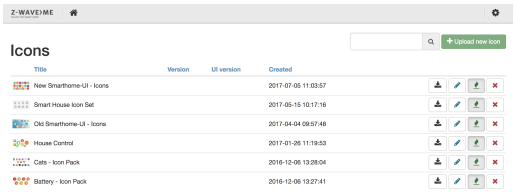


Figure 10.6: Select the Icon pack

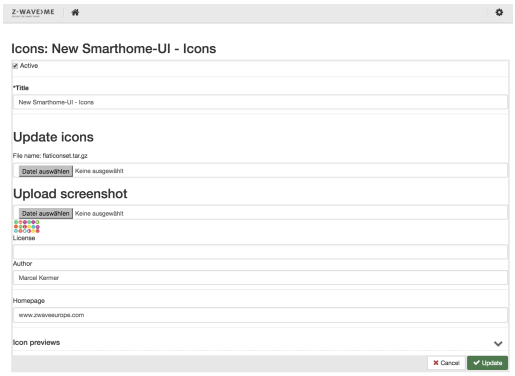


Figure 10.7: Manage an Icon pack

The total size of the archive must not exceed 2 MB.

10.2.3 Upload your Icon Set

For this please register at

http://developer.z-wave.me/.

Once logged in, the right-hand side menu icon allows opening the icon set management dialog as shown in Figure 10.6

Here you can create a new Icon pack record and manage you existing ones:

Please choose the title of your icon pack and provide a screenshot. This is the image shown in the icon set preview as described in Section 4.2.3.

10.3 How to translate the Z-Way to your language

Z-Way operates with two different user interface both having its own translation engines and translation files. Additionally, the backend code will translate certain tokens already when handling the data. Finally, the applications from the app store may need to be translated as well.

You can call all relevant files from info@z-wave.me or download from GitHub

https://github.com/Z-Wave-Me/zwave-smarthome/tree/dev.

You can also use the installation of Z-Way on your system if you have access to the file system. The following tutorial assumes you have this access. This also allows you to translate the strings and subsequently test the new UI. All Z-Way code you find in the folder and subfolders of /z-way-server.

Please note that you can include your name, email, and company web page link to both UIs as a reward for doing the work.

10.3.1 Smart-Home User Interface

All translation tokens for the Smart Home UI can be found in /htdocs/smarthome/app/lang. Just copy the file en.json to XX.json with XX as ISO 2 char code of your language. Start translating the new file into your language.

In app/config.js find a language list array “lang_list”: [’en’, ’de’, ’ru’, ’cn’, ’fr’]. Just add your 2-char code. In / htdocs /smarthome/app/images/flags the flag of your country/language needs to be added. The code expects the file name XX.png with a 24x24 pixel PNG image.

For the standardized form dialogs of the application setups you find the language tokens in app/core/config.js. The array “lang_codes” need to be extended by your language.

10.3.2 Expert User Interface

All translation tokens for the Smart Home UI can be found in /htdocs/expert/app/lang. Just copy the file en.json to XX.json with XX as ISO 2 char code of your language. Start translating the new file into your language.
In app/config.js find a language list array “lang_list”: [’en’, ’de’, ’ru’, ’cn’, ’fr’]. Just add your 2-char code.

10.3.3 Backend Code

The subfolder /translations contains some XML files that are required for backend actions and rendering. These renderings are done in the backend and used in both Browser Type User Interfaces. The main files to extend with own language are:

1. ScaleIDs.xml: Contains the scales for multilevel sensors. These Scale Strings are displayed on both Expert UI and Smart Home UI whenever a sensor value is shown
2. Alarms.xml: Contains the Strings for the various Alarm types of Z-Wave. These strings are used in Expert UI and for the initial device name in Smart Home UI.
3. ColorCapabilities.xml: Contains the types of color settings. This is used for the initial device name in Z-Way Smart Home Interface .

To alter and extend these files, please find out the ISO 2 char code of your language and add one line for each token.

10.3.4 Submission of your Language Pack

Please send the translated language files XX.json plus your flag plus any changed XML file zipped to info@z-wave.me for inclusion into the next release of the software.

11 Develop Code for Z-Way

11.1 Z-Way software structure overview

Z-Way offers multiple Application Program Interfaces (API) that are partly built on each other. Figure 11.1 shows the general structure of Z-Way with focus on the APIs. The most important part of Z-Way is the Z-Wave core. The Z-Wave core uses the standard Silicon Labs Serial API to communicate with a Z-Wave compatible transceiver hardware but enhanced with some Z-Way specific functions such as frequency change. The standard interface is not public but available for owners of the Silicon Labs Development Kit (SDK) [1]

The Z-Wave core services can be accessed directly using the Z-Wave Device API (zDev API). There are two Z-Wave device API versions available:

- Z-Wave API as JSON API: All functions are available using a JSON API implemented by an embedded webserver. This web server can be used in two ways:
  - web sockets, a permanent IP connection
  - REST (Representational State Transfer)

  Both ways to use the JSON API have the same data structures and commands. The **Z-Wave Expert User Interface** as described in chapter 7 uses the REST option of this JSON API. This user interface is a very good reference how to apply the Z-Wave Device API. For oore information about the Z-Wave Device API please refer to Section 11.2.1.
- Z-Wave API as C Library API: All functions of the JSON API are available as C library function too. The URL

  `razberry.z-wave.me/fileadmin/z-way-test.tgz`

  provides a sample application written in standard C that makes use of the C level API to demonstrate its application. Makefiles and project files for compilation on Linux and OSX are provided together with the sample code. More information on the C level API you find in Section 11.4.

The **Z-Wave device API only allows the management of the Z-Wave network** and the control and management of the devices as such. No higher order logic except the so-called Z-Wave associations between two Z-Wave devices can be used.

For all **automation and higher order logic a JavaScript automation engine** is available. This engine is also shipped with Z-Way.

The JavaScript API on top of the JS engine mirrors all functions of the Z-Wave Device API but also allows access to third-party device APIs (e.g. EnOcean). This means the JS API is the common ground for all further application logic and user interfaces (with the exception of the **Z-Wave Expert User Interface** that uses the Z-Wave Device API.

The JavaScript layer makes use of a JavaScript implementation provided by Google it is also used in Googles Chrome web browser. All JavaScript API functions can also be accessed using the embedded webserver. The beauty of this interface is that JavaScript can be executed on the server and on the client. Executing on the client makes sense for small changes to the data model or running small helper programs.

There are two important sub-portions of the JavaScript layer:

- Virtual Devices (vDevs): All functions of the physical devices plus other functions are mapped into virtual devices. Virtual devices have properties and attributes linked to their physical counterpart functions. The Z-Way Smart Home Interface is completely written using the virtual device concept and this user interface can act as a good reference how to use them. Please refer to chapter 11.3 for more information about the use of the vdev concept.
- The Apps: These are JavaScript portions that are dynamically loaded into the JS core and implement application or user specific function. Please refer to chapter 6 for more information about the app concept and existing apps. Please refer to Chapter 13.2 for more information about how to develop own apps.

11.2 Z-Way APIs Quick Reference

11.2.1 Z-Wave Device API

The Z-Wave Device API implements the direct access to the Z-Wave network. All Z-Wave devices are referred to by their unique identification in the wireless network—the Node ID. Z-Wave devices may have different instances of the

[1] The Silicon Labs SDK is available from Arrow (www.arrow.com). Depending on the hardware options chosen the price varies between 2000 and 4000 USD only.

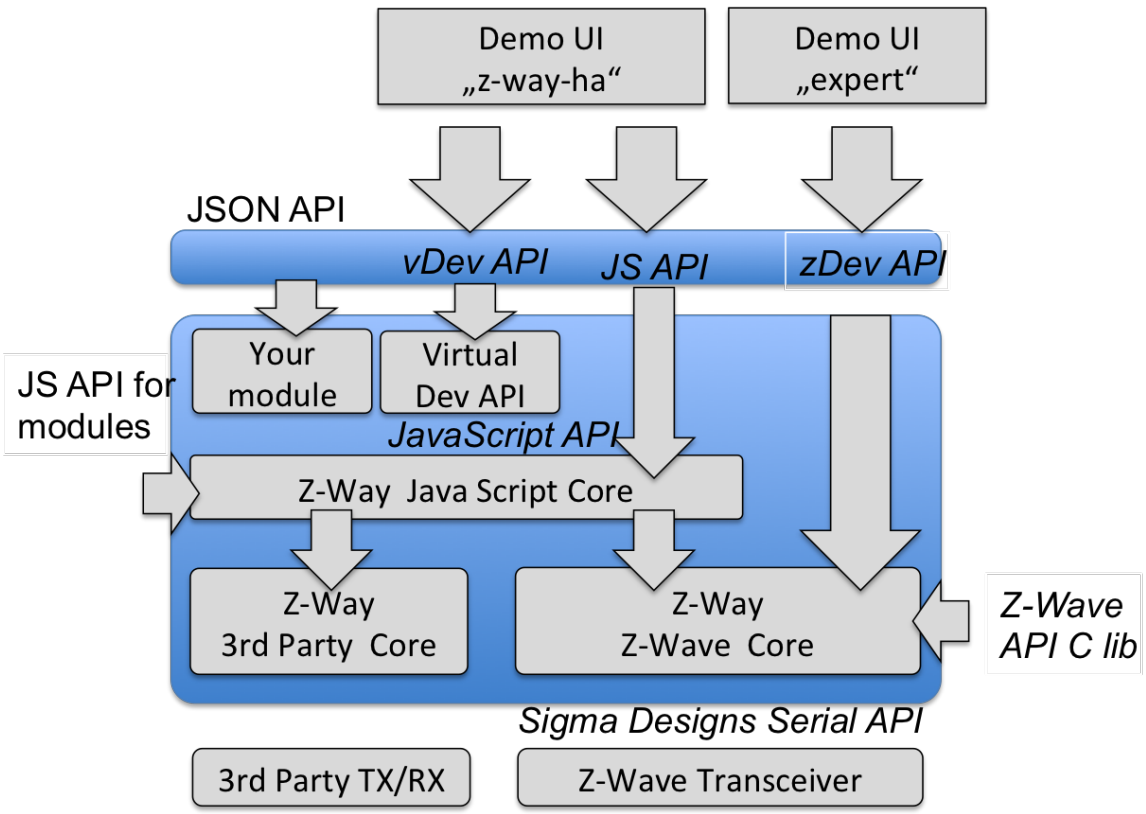


Figure 11.1: Z-Way APIs and their use by GUI demos

same function, also called channels (for example sockets in a power strip). The Z-Wave Device API refers to them as daughter objects of the physical device object identified by an instance ID. In case there is only one instance, the instance ID = 0 is used.

Sending Z-Wave Commands

All device variables and available commands in a Z-Wave device are grouped in the so-called command classes. The Z-Wave API allows direct access to all parameters, values and commands of these command class structures. Annex B gives you the complete reference of the implemented command classes.
Beside the devices the Z-Wave Device API also offers access to the management interface of the network. Annex C gives you a full reference of the implemented function classes.
The Z-Wave Device API can be accessed on the JSON API with any standard web browser using the URL

```
http://YOURIP:8083/ZWaveAPI/*
```

.
Device objects or commands of these objects are accessed by

```
http://YOURIP:8083/ZWaveAPI/Run/devices[*].*
http://YOURIP:8083/ZWaveAPI/Run/devices[x].instances[y].*
http://YOURIP:8083/ZWaveAPI/Run/devices[x].instances[y].commandClasses[z]
.*
```

The whole data tree of the Z-Wave network is accessed using

```
http://YOURIP:8083/ZWaveAPI/Data/*
```

.
Please refer to the Z-Way for information about the context, the commands, and the data used. Section 11.3 provides more information about the API and the underlying data structure.
All acces ot the webserver require authentication of the user. Please refer to Chapter 13.1 for details how to authenticate.

11.2.2 JavaScript API (JS API)

The Z-Wave Device API or any other third-party technology API do not offer any higher order logic support but the pure access to functions and parameters of devices only.

Z-Way offers an automation engine to overcome this restriction. A server-side JavaScript runtime environment allows writing JavaScript modules that are executed within Z-Way (means on the server). The same time all functions of the JS API can also be accessed on the client side (the web browser). This offers some cool debug and test capabilities. Among others it is possible to write whole JS functions right into the URL or the browser.

The JS API can be accessed from the web browser with the URL

```
http://YOURIP:8083/JS/Run/*
```

Among others the whole Z-Wave Device API is available within the JS API using the object "zway''. As a result, the following three statements refer to the very same function:

1. **http://YOURIP:8083/ZWaveAPI/Run/devices[3].*** Client Side URL access using the Z-Wave Device API.
2. **http://YOURIP:8083/JS/Run/zway.devices[3].***: Client Side URL access using the JS API
3. **zway.devices[3].***: Server Side access using the JS and the public zway object

Due to the scripting nature of JavaScript it is possible to "inject'' code at run time using the interface. Here a nice example how to use the JavaScript setInterval function:

Listing 1. Polling of device #2

```
/JS/Run/setInterval(function() {
    zway.devices[2].Basic.Get();
}, 300*1000);
```

This code will, once "executed'' as URL within a web browser, call the Get() command of the command class Basic of Node ID 2 every 300 seconds.

A very powerful function of the JS API is the ability to bind functions to certain values of the device tree. They get then executed when the value changes. Here is an example for this binding. The device No. 3 has a command class SensorMultilevel that offers the variable "level." The following call—both available on the client side and on the server side—will bind a simple alert function to the change of the variable.

Listing 2. Bind a function

```
zway.devices[3].SensorMultilevel.data[1].val.bind(
function() {
debugPrint('CHANGED TO:' + this.value + '\n');
});
```

[2]

Chapter 11.3 and 11.3.1 describe the whole JS API in detail. The names and IDs of the different command classes as well as their instance variables can be found in the Annex B.

JavaScript modules can and will generate new functions that are accessible using the JSON interface. For simplification function calls on the API (means on the client side) are written in URL style starting with the word "ZAutomation":

/ZAutomation/JSfunction/JParameter == JSfunction(JParameter)

11.2.3 Virtual Device API

All functions and instances of a physical device, which are represented as daughter objects in the Z-Wave Device API, are enrolled into individual virtual devices.

In case the Z-Wave API shows one single physical device with two channels, the Virtual Device API will show two devices with similar functionality. In case the Z-Wave API shows a physical device with several functions (like a binary switch and an analog sensor in one device), the Virtual Device API (vDev API) will show them as several devices with one function each.

The vDev is accessed using the JSON API in a slightly different style than zDev API. All devices, variables, and commands are encoded into a URL style for easier handling in AJAX code. A typical client-side command in the vDev API looks like

```
http://YOURIP:8083/ZAutomation/api/v1/devices/ZWayVDev_6:0:37/command/off
```

[2] Please note that the Sensor Multilevel Command class data is an array index by the scale ID. Other command classes such as Basic do not have this index but allow direct access using CommandClassName.data.level

API Type	Core Function	Network Management	Automation
Z-Wave Dev API (JSON)	Access to physical network and physical devices via JSON	Yes	No
Z-Wave Dev API (C lib)	Access to physical network and physical devices via C style calls	Yes	No
JavaScript API	Access to physical network and devices plus JS type functions	No	Yes, via zDev
vDev API	Unified Access to functions of devices, optimized for AJAX GUI	No	Yes

Table 11.1: Different APIs of the Z-Way system

"API" points to the vDev API function, "v1" is just a constant to allow future extensions. The devices are referred to by a name that is automatically generated from the Z-Wave Device API. The vDev also unifies the commands "command" and the parameters, here "off."
On the server side, the very same command would be encoded in a JavaScript style.

Listing 3. Bind a function

```
dev = this.controller.devices.get('ZWayVDev\_6:0:37');
dev.command('off');
```

The vDev API also offers support for notifications, locations information, the use of other modules, etc.

11.2.4 Comparison

Table 11.1 summarizes the functions of the different APIs.

11.3 The Z-Wave Device (JSON) API in detail

This chapter describes the Z-Wave Device API and its use in detail All examples will use the HTTP/JSON API notation. Please note that the C library notation offers equal functionality.

The Z-Wave Device API is the north-bound interface of the Z-Wave Core. This Z-Wave core implement the whole control logic of the Z-Wave network. The two main functions are

- Management of the network. This includes including and excluding devices, managing the routing and rerouting of the network an executing some housekeeping functions to keep the network clean and stable. The function classes can be seen as functions offered by the controller itself. Hence the variables and status parameters of the networks are offered by an object called "controller."
- Execution of commands offered by the wireless devices as such switching switches and dimming dimmers. Z-Wave groups the command and their corresponding variables into so-called command classes. The Z-Wave API offers access to these command classes with their variables and their commands according to the abilities of the respective device.

The description of function classes and command Cclasses and their access using the JSON API complete the description of the Z-Wave Device API. For a full reference of function classes and command classes please refer to the Annex C and B.

11.3.1 The data model

Z-Way holds all data of the Z-Way network in a data holder structure. The data holder structure is a hierarchical tree of data elements.

Following the object-oriented software paradigm the different commands targeting the network or individual devices are also embedded into the data objects as object methods.

Each data element is handled in a data object that contains the data element and some contextual data.

The Data object

Each Data element such as devices[nodeID].data.nodeId is an object with the following child elements:

- value: the value itself
- name: the name of the data object
- updateTime: timestamp of the last update of this particular value
- invalidateTime: timestamp when the value was invalidated by issuing a Get command to a device and expecting a Report command from the device

Every time a command is issued that will have impact on a certain data holder value the time of the request is stored in "invalidateTime". This allows tracking when a new data value is requested from the network and when this new data value is provided by the network.

This is particularly true if Z-Way is sending a SET command. In this case the data value is invalidated with the "SET" commands and gets validated back when the result of the GET command was finally stored in the data model.

To maintain compatibility with JavaScript the data object has the following methods implemented:

- valueOf(): this allows to obmit .value in JS code, hence write as an example data.level = 255
- updated(): alias to updateTime
- invalidated(): alias to invalidateTime

These aliases are not enumerated if the dataholder is requested (data.level returns value: 255, name: "level", updatedTime: 12345678, invalidatedTime: 12345678).

The Data and Method Tree

The root of the data tree has two important child objects:

- controller, this is the data object that holds all data and methods (commands, mainly function classes) related to the Z-Way controller as such
- devices array, this is the object array that holds the device -specific data and methods (commands, mainly command classes).

11.3.2 Timing behavior of Z-Wave data

Please note that all status variables accessible on the Z-Wave Device APIs are only proxy of the real value in the network.

To transport data between the real wireless device and the GUI multiple communication instances are involved. The complexity of this communication chain will be explained in the following example:

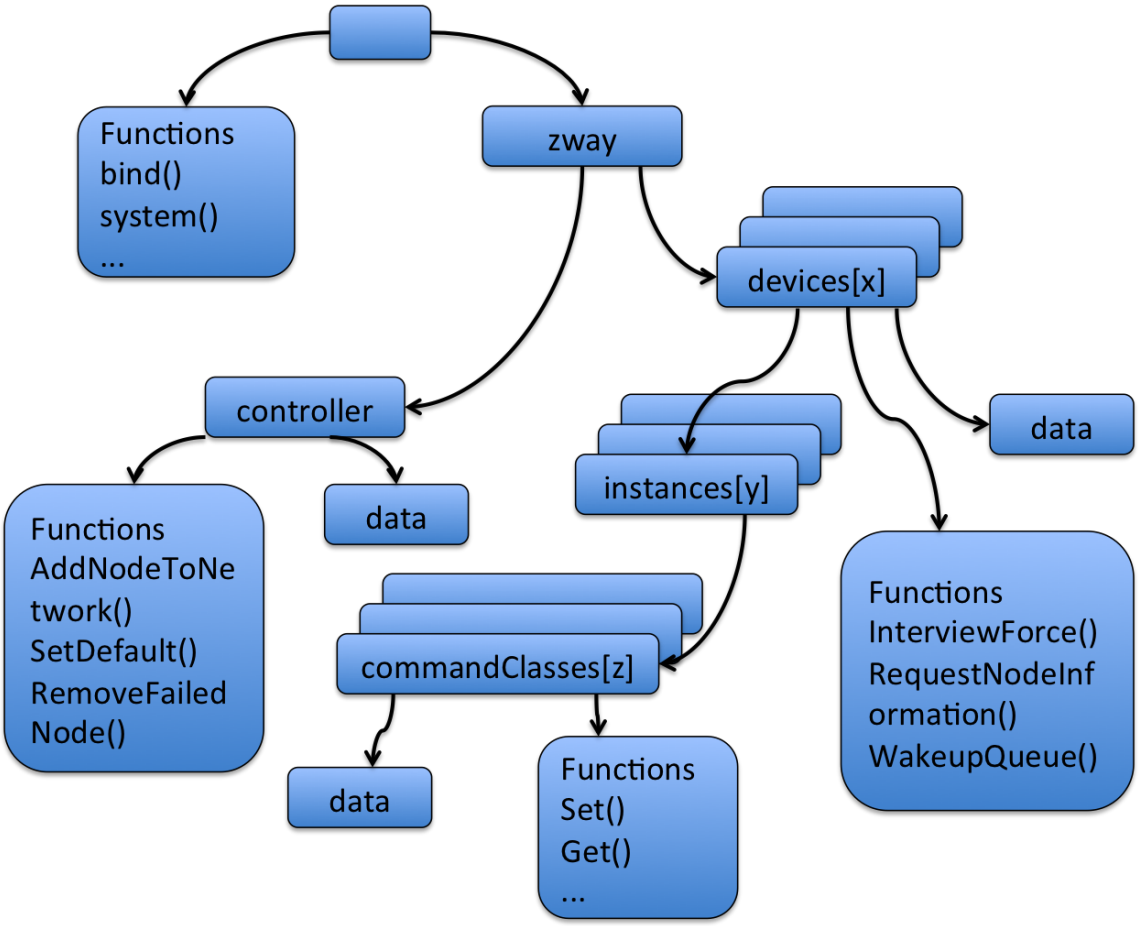


Figure 11.2: Z-Way Object Tree Structure

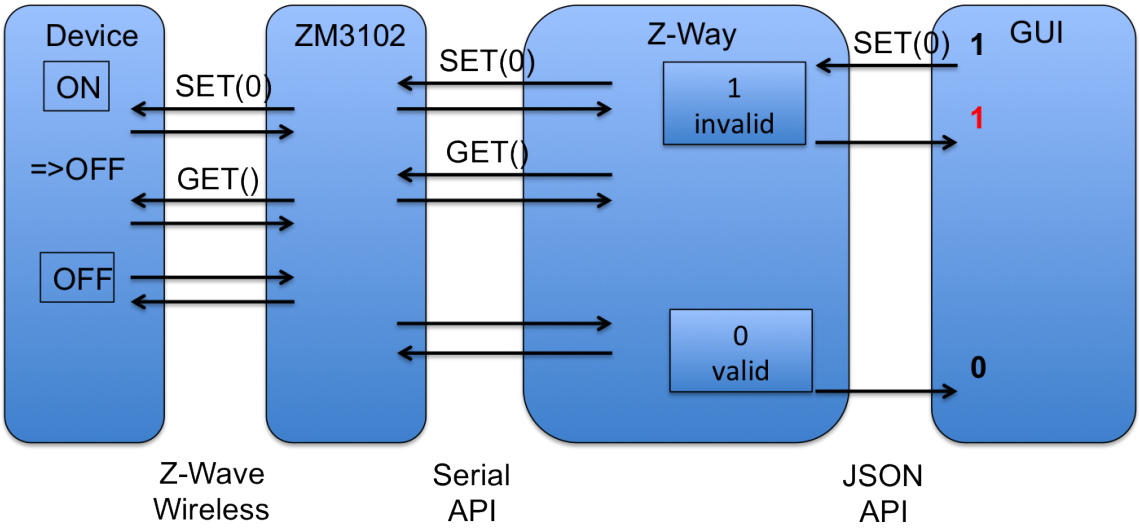


Figure 11.3: Z-Way Timings

Assuming the GUI shows the status of a remote switch and allows changing the switching state of this device. When the user hits the switching button, he expects to see the result of his action as a changing status of the device in the GUI. The first step is to hand over the command (SET) from the GUI to Z-Way using the JSON interface. Z-Way receives the command and will confirm the reception to the GUI. Z-Way recognizes that the execution of the switching command will likely result in a change of the status variable However Z-Way will not immediately change the status variable but invalidate the actual value (mark as outdated). This is the correct action because at the moment when the command was received the status on the remote device has not been changed yet but the status of the switch is now unknown. If the GUI polls the value it will still see the old value but marked as invalid. Z-Way will now hand over the switching command to the Z-Wave transceiver chip. Since it is possible that there are other command waiting for execution (sending) by the Z-Wave transceiver chip the job queue is queuing them and will handle certain priorities if needed. Z-Way has recognized that the command will likely change the status of the remote device and is therefore adding another command to call the actual status after the switching command was issued. The transceiver is confirming the reception of the command and this confirmation is noted in the job queue. This confirmation however only means that the transceiver (Z-Wave chip) has accepted the command and does neither indicate that the remote device has receives it nor even confirming that the remote device has executed accordingly. The transceiver will now try to send the command wirelessly to the remote device. A successful confirmation of the reception from the remote device is the only valid indicator that the remote device has received the command (again, not that it was executed!). The second command (GET) is now transmitted the very same way and confirmed by the remote device. This device will now sent a REPORT command back to Z-Way reporting the new status of the switching device. Now the transceiver has to confirm the reception. The transceiver will then send the new value to the Z-Way engine by issuing commands via the serial interface. Z-Way receives the report and will update the switching state and validate the value. From now on the GUI will receive a new state when polling.

11.3.3 Executing Commands

JSON API allows executing commands on the server side using HTTP POST or GET requests. The command to execute is taken from the URL.
All functions are executed in form

```
http://YOURIP:8083/Run/ZWaveAPI*
```

.
The best way to learn about the commands and the data is to use the **Z-Wave Expert User Interface** plus a JavaScript Debugger to see the command the AJAX code of the **Z-Wave Expert User Interface** sends to the Z-Way server backend. Additionally the Z-Wave Expert User Interface provides nice and convenient vizualization of all commends (both command classes and function classes).
All acces ot the webserver require authentication of the user. Please refer to Chapter 13.1 for details how to authenticate.

Function Class Commands

Figure 11.4 shows the Controller Info Page in Network menu with a list of all function classes implemented. The complete reference of the parameters and return values of the functions classes you find in annex C.
Assuming there is a function class “SerialAPIGetInitData” it is possible to call the function by calling the URL

```
/ZWaveAPI/Run/SerialAPIGetInitData(0)
```

in the web browser. In case the function was completed successfully, a simple “null” is returned; otherwise, an error code is provided.

Device Command Class Commands

In the same manner, it is possible to send a command to a device using one of its command classes. The **Z-Wave Expert User Interface** provides a general menu item called Expert Commands as shown in Figure 11.5. Z-Way reads out all command classes and its functions and provides here a complete list of command lass-specific commands. The debug window will reveal the syntax if the complete command class reference in Annex B is not available or too inconvenient to use.
For example, to switch ON a device no 2 using the command class BASIC, it is possible to write:

```
/ZWaveAPI/Run/devices[2].instances[0].commandClasses[0x20].Set(255)
```

or

```
/ZWaveAPI/Run/devices[2].instances[0].Basic.Set(255)
```

The **Z-Wave Expert User Interface** has a JavaScript command

```
runCmd(<command>)
```

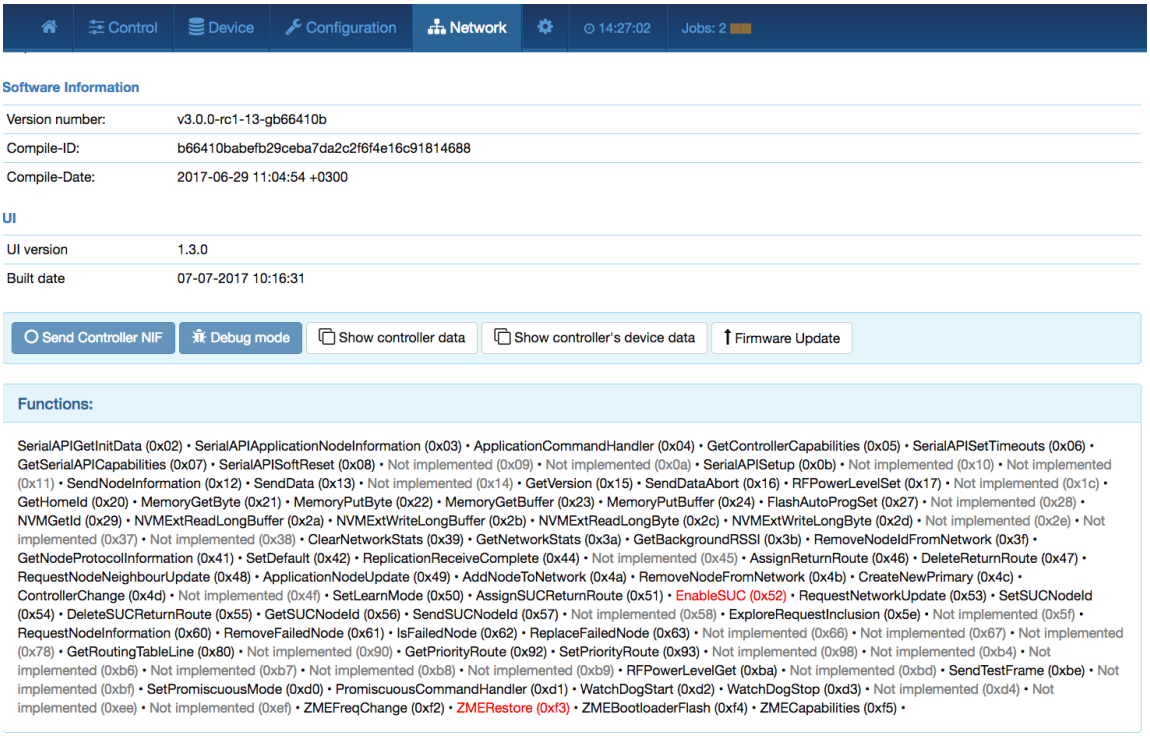


Figure 11.4: Z-Way Function Classes

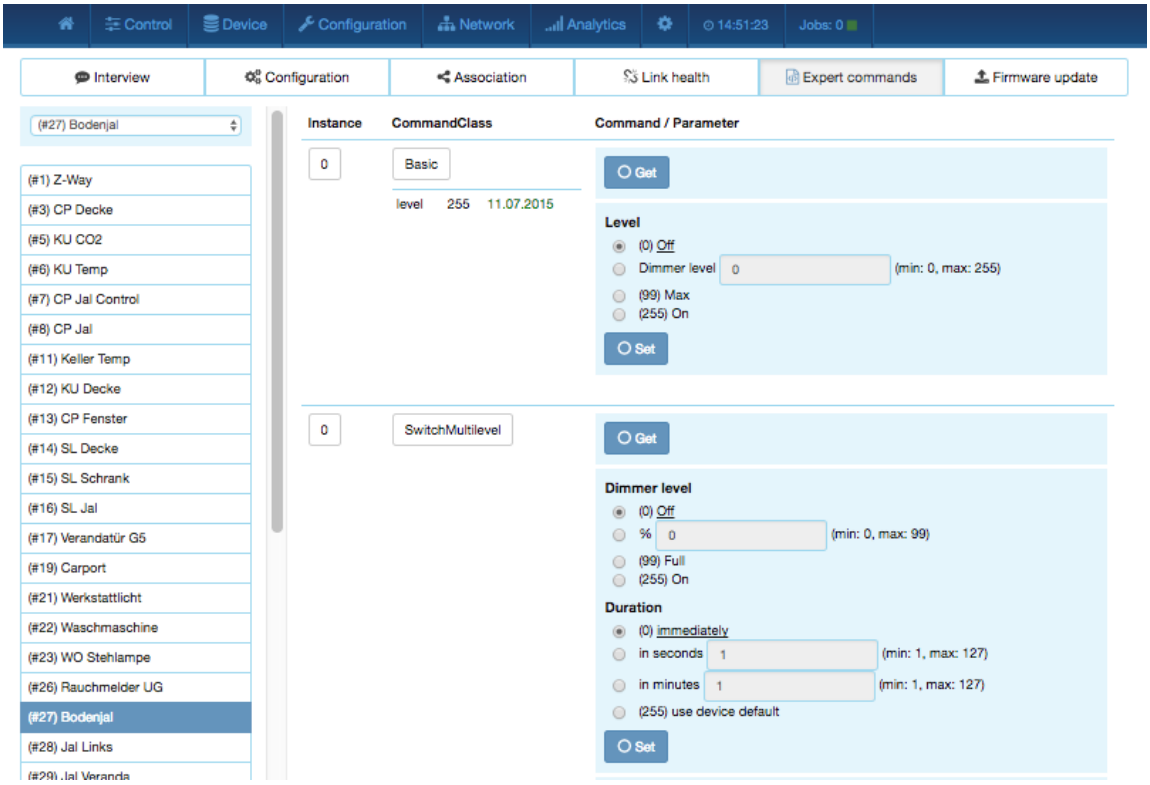


Figure 11.5: Z-Way Expert Command Class Commands

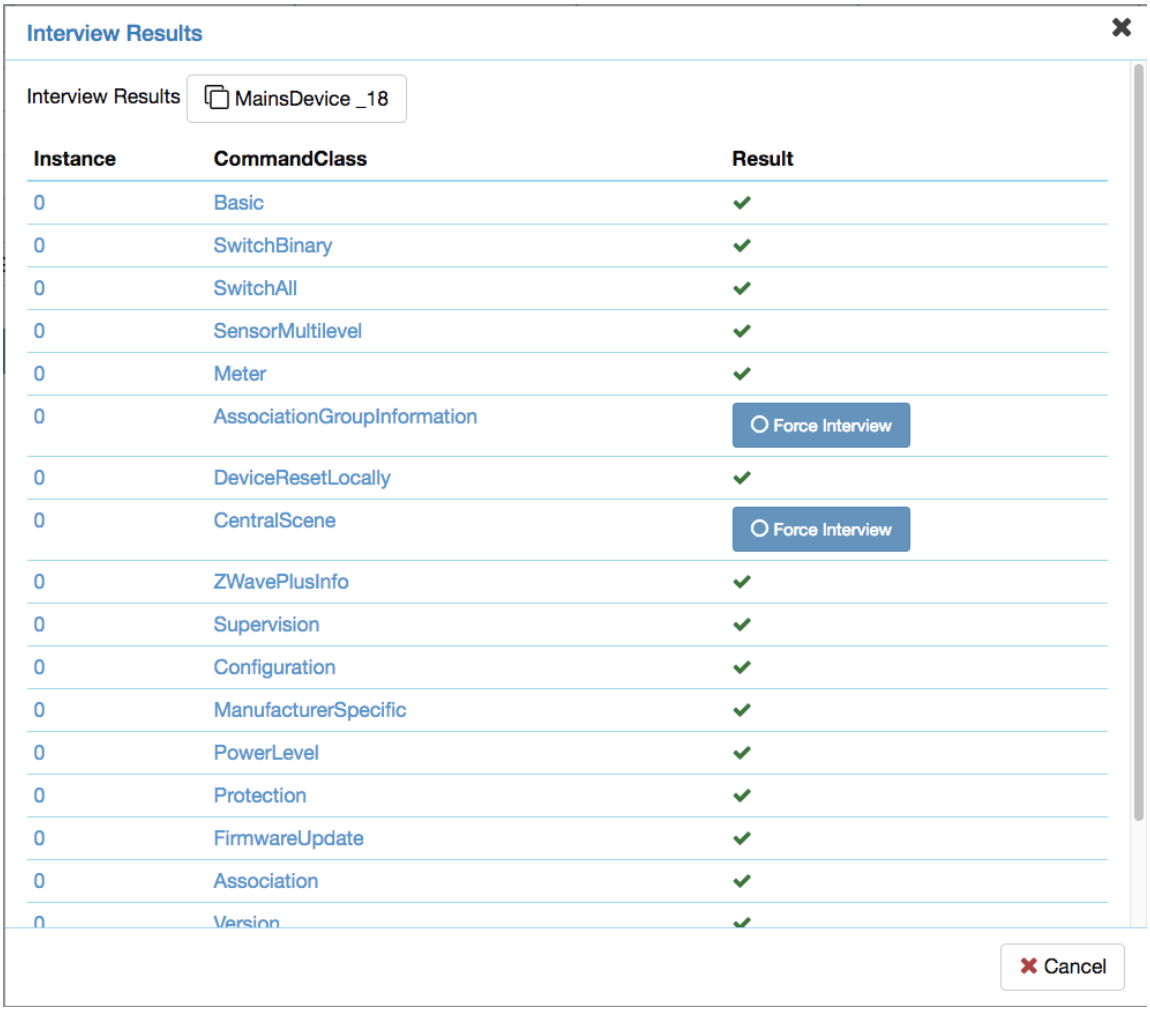


Figure 11.6: Command Class Inerview overview

to simplify such operations. This function is accessible in the JavaScript console of your web browser (in Chrome you find the JavaScript console under View->Debug->JS Console). Using this feature, the command in JS console would look like

```
runCmd('devices[2].instances[0].Basic.Set(255)')
```

The usual way to access a command class is using the format 'devices[nodeId].instances[instanceId]. commandClasses[commandclassId]'. There are ways to simplify the syntax:

- "devices[nodeId].instances[instanceId].Basic" is equivalent to "devices[nodeId].instances[instanceId].commandClasses[0x20]"
- the instances[0] can be obmitted: "devices[nodeId].instances[instanceId].Basic" then turns into "devices[nodeId].Basic"

Accessing Data

The data model or data holder object as described is Section 11.3.1 can be accessed completely using the **Z-Wave Expert User Interface** . The two buttons Show controller Data and Show controllers device data in Network > Controller Info of **Z-Wave Expert User Interface** as shown in Figure 11.4 lists all variables of the controller as such. One structure is controller-specific and one other structure is the data of the controller as node of the Z-Wave network. All nodes of the Z-Wave network have the very same data structure beside their individual array of instances and command classes per instance. This data model for the individual devices can be access using Configuration > Show Interview results in **Z-Wave Expert User Interface** . Figure 11.6 shows this dialog. On the top of the window there is a button with the devices name. This button reveals the data structure of the individual device as shown in Figure 11.7.
The dialog has the list of all command classes and clicking on the name of the command class will open a sub dialog showing the data of the commend class. Each command class has some permanent values:

- supported: This indicates if this command classes is supported or controlled only
- version: This is the version number of the command class as detected during the device interview using the

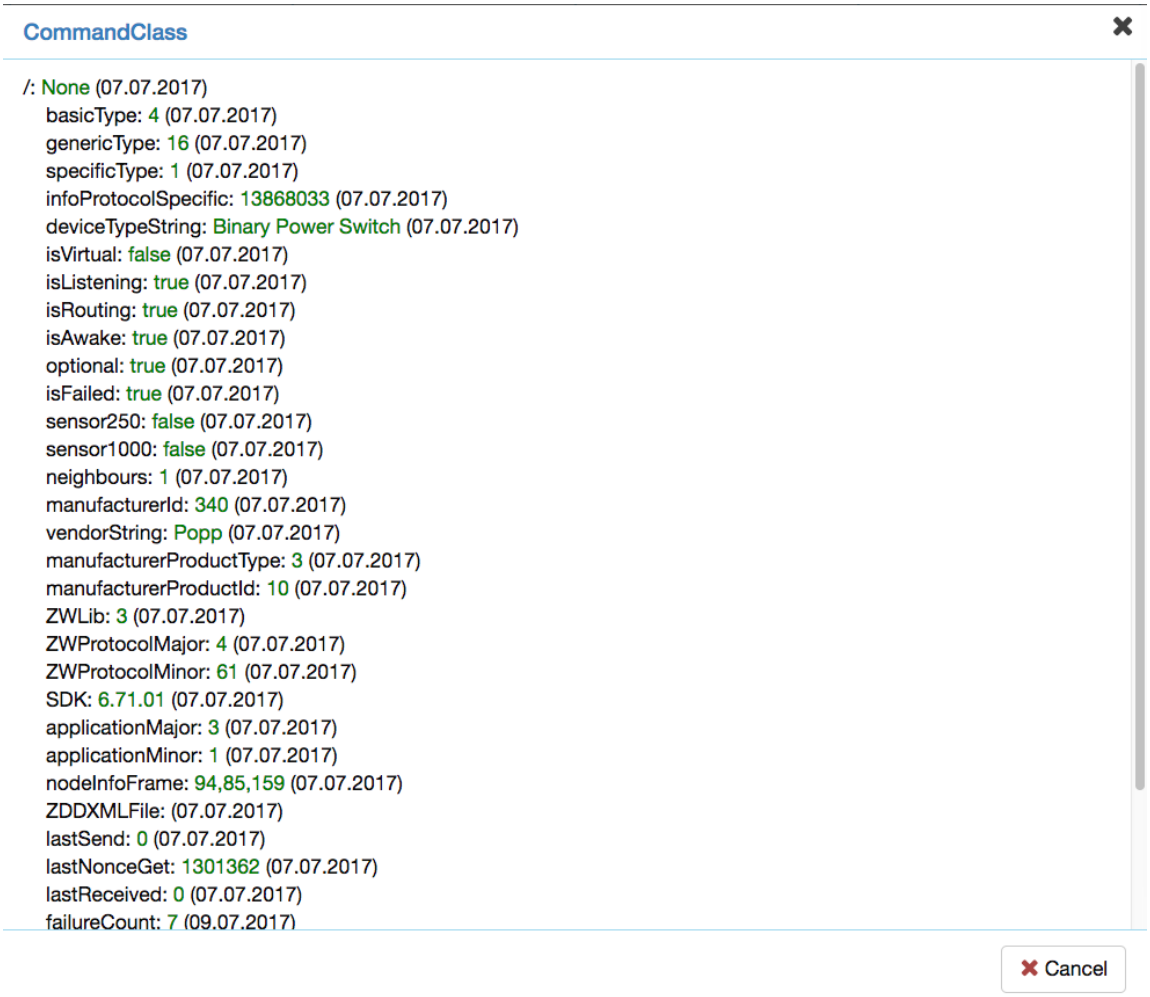


Figure 11.7: Command Class Variables in **Z-Wave Expert User Interface**

command class “VERSION”
- security: Indicates if this command class is within the security environment
- interviewDone: This flag indicates if the interview of this particular command classes passed
- interviewCounter: This is a helper variable that is counted down on every attempt to interview the devices. Its default value is 10. If it reaches 0 (10 unsuccessful attempts), Z-Way will give up interviewing. This makes sure that Z-Way is not blocked by devices with wrong implementation not passing interview.

Any data holder object has properties value, updateTime, invalidateTime, name, but for compatibility with JS and previous versions we have valueOf() method (allows omitting .value in JS code, hence write "data.level == 255"), updated (alias to updateTime), invalidated (alias to invalidateTime).

/ZWaveAPI/Data/<timestamp>

Returns an associative array of changes in Z-Way data tree since <timestamp>. The array consists of (<path>: <JSON object>) object pairs. The client is supposed to assign the new <JSON object> to the subtree with the <path> discarding previous content of that subtree. Zero (0) can be used instead of <timestamp> to obtain the full Z-Way data tree.

The tree have same structure as the backend tree (Figure 11.2) with one additional root element "updateTime" which contains the time of latest update. This "updateTime" value should be used in the next request for changes. All timestamps (including updateTime) corresponds to server local time.

The object looks like:

Listing 1. JSON Data Structure

```
\{
  "[path from the root]": [updated subtree],
  "[path from the root]": [updated subtree],
  ...
  updateTime: [current timestamp]
}
```

Examples for Commands to update the data tree look like:

Get all data: /ZWaveAPI/Data/0

Get updates since 134500000 (Unix timestamp): /ZWaveAPI/Data/134500000

Please note that during data updates some values are updated by big subtrees. For example, in Meter Command Class value of a scale is always updated as a scale subtree by [scale].val object (containing scale and type descriptions).

/ZWaveAPI/InspectQueue

This function is used to visualize the Z-Way job queue. This is for debugging only but very useful to understand the current state of Z-Way engine.

The information given on this page is only relevant for advanced Z-Wave developers and for debugging.

The table shows the active jobs with their respective status and additional information.

Table 11.2 summarizes the different values displayed on the Job Queue visualization. While this info is certainly not relevant for end users of the system it is a great debug tool.

Handling of updates coming from Z-Way

A good design of a user interface is linking UI objects (label, textbox, slider, ...) to a certain path in the tree object. Any update of a subtree linked to user interface will then update the user interface too. This is called bindings.

For web applications Z-Way contains a library called jQuery.triggerPath (extention of jQuery written by Z-Wave.Me), that allows making such links between objects in the tree and HTML DOM objects. Use

var tree;

jQuery.triggerPath.init(tree);

during web application initialization to attach the library to a tree object. Then run

jQuery([objects selector]).bindPath([path with regexp], [updater function], [additional arguments]);

to make binding between path changes and updater function. The updater function would be called upon changes in the desired object with this pointing to the DOM object itself, first argument pointing to the updated object in the tree, second argument is the exact path of this object (fulfilling the regexp) and all other arguments copies additional arguments. RegExp allows only few control characters: * is a wildcard, (1|2|3) - is 1 or 2 or 3.

n	This column shows the number of sending attempts for a specific job. Z-Way tries three times to dispatch a job to the transceiver.
W,S,D:	This shows the status of the job. If no indicator is shown the job is in active state. This means that the controller just tries to execute the job. “W” states indicated that the controller believes that the target device of this job is in deep sleep state. Jobs in “W” state will remain in the queue to the moment when the target device announces its wakeup state by sending a wakeup notification to the controller. Jobs in “S” state remain in the waiting queue to the moment the security token for this secured information exchanged was validated. “D” marks a job as done. The job will remain in the queue for information purposes until a job garbage collection removed it from the queue.
ACK:	shows if the Z-Wave transceiver has issued an ACK message to confirm that the message was successfully received by the transceiver. This ACK however does not confirm that the message was delivered successfully. A successful delivery of a message will result in a “D” state of this particular job. If the ACK field is blank, then no ACK is expected. A “.” indicates that the controller expects an ACK but the ACK was not received yet. A “+” indicates that an ACK was expected and was received.
RESP	shows if a certain command was confirmed with a valid response. Commands are either answered by a response or a callback. If the RESP field is blank, then no response is expected. A “.” indicates that the controller expects a response, but the response has not been received yet. A “+” indicates that a response was expected and has been received.
Cbk	If the Cbk field is blank, then no callback is expected. A “.” indicates that the controller expects a Callback but the Callback was not received yet. A “+” indicates that a Callback was expected and was received.
Timeout	Shows the time left until the job is de queued
Node Id	shows the ID of the target node. Communication concerning the network, like inclusion of new nodes, will have the controller nodeID as a target node ID. For command classes command the node ID of the destination Node is shown. For commands directed to control the network layer of the protocol, the nodeID is zero.
Description	shows a verbal description of the job
Progress	shows a success or error message depending on the delivery status of the message. Since Z-Way tries three times to deliver a job up to 3 failure messages may appear. Buffer: It shows the hex values of the command sent within this job

Table 11.2: Parameters of the Job Queue Vizualization

Please note that the use of the triggerpath extension is one option to handle the incoming data. You can also extract all the interesting values right when the data is received and bind update functions to them.

Handling S2 inclusion process

Security S2 inclusion process require additional interaction with the user during inclusion process. In addition to putting the device in Learn Mode and Z-Way in Add mode the user will be asked to grant different Security S2 keys and enter the PIN code.
To implement Security S2 inclusion process in your own app follow the steps below:

- Include the device
- Check if it supports S2 (if the Command Class exists)
- If Smart Start is in progress, nothing to be done — Z-Way will handle everything automatically
- Wait for `devices[N].instances[0].commandClasses[159].data.requestedKeys` to become True
- Check `data.requestedKeys` children and set `data.grantedKeys` children accordingly. In most cases you would just copy from requested to granted (grant all keys the device requested for).
- Set `data.grantedKeys` to True to notify Z-Way that you are done with keys selection
- Wait for `data.publicKeyAuthenticationRequired` to be set. If False (device was not granted Access or Authenticated keys), copy in `data.publicKeyVerified` the `data.publicKey` (it is recommended to show first two bytes in decimal form to the user to confirm). Otherwise copy and fill first two bytes with the PIN code.
- A failed Security S2 interview will result in `data.securityAbandoned` to be set to True. A successfull one will have `data.securityAbandoned` equal to False and `data.interviewDone` equal to True.

11.4 C-Library API and a general view on the Z-Way file structure

11.4.1 Files in the /zway folder

Z-Way keeps all files in one folder with exception of the log files. In Unix based platforms such as Linux PC, Raspberry Pi or open WRT the standard install folder is usually **/opt/zway-server**
The logfile is typically placed in **/var/log/z-way-server.log**
but this location of the log file can be reconfigured in the config file.
On Windows, the installation wizard asked where to place Z-Way and where to place the log file.
Right after installation, the standard folder has the following content:

```
z-way-server
 ├─ automation:  The JavaScript sub system
 ├─ config:  Various configuration xml files
 ├─ htdocs:  The web servers doc directory
 ├─ libs:  The binary libs such as libzway etc.
 ├─ libzway Header files for libzway and other libs
 ├─ modules:  binary modules
 ├─ modules-includes:  header file for binary modules
 ├─ translations:  XMLs mapping Ids to text
 ├─ ZDDX: Device Description Files
 ├─ z-get-tty-config:  Config for utility
 ├─ ChangeLog
 ├─ config.xml:  Main config file
 ├─ z-cfg-update:  Utility
 ├─ z-get-tty:  Utility
 └─ z-way-server:  Main Application
```

config.xml - the main config file

The main config file in the root folder has XML file format. If only allows setting the log level (0 = log all, 9 = log almost nothing), the path to the log file and a debug port if needed. Don't change the setting for automation folder unless you really know what you do and why.
This is an example for the standard config.xml displaying all log right into the console.

Listing 1. config.xml

```
<config>
    <automation-dir>automation</automation-dir>
    <log-file></log-file>
    <log-level>0</log-level>
    <debug-port>0</debug-port>
</config>
```

config: Various configuration xml files

This subfolder has the following structure:

```
config
 ├─ maps:  legacy folder
 ├─ zddx
 │   └─ HOMEID-DeviceData.xml:  main storage for Z-Wave structure
 ├─ Defaults.xml:  Default Settings
 ├─ Profiles.xml
 └─ Rules.xml
```

The file **Defaults.xml** allows defining various behavior of Z-Way, among them the appearance of Z-Way as secondary controller:

- AutoConfig: Flag if Z-Way shall interview the device right after inclusion (default = 1)
- DeepInterview: Flag that Interview is only completed after all values are received back. This includes asking the device for all initial values of sensor or status data (default = 1)
- SaveDataAfterInterviewSteps: Flag whether or not all device data will be saved after each interview step (default = 1)
- TryToBecomeSIS: Will Z-Way try to become networks SIS if transceiver hardware allows it to (default = 1)?
- SecureInterviewAsInclusionController: Z-Way will initiate interview as Inclusion Controller, if 0 - only as Primary/SIS
- SecureInterviewAcceptedWithoutSchemeInherit: If 1 - Z-Way will not fail secure interview as secondary/inclusion controller if Scheme Inherit is not received, if 0 - fail interview as by Z-Wave protocol
- SecureAllCCs: Always use Security if possible (even for CCs allowed as non-secure)
- DeviceReplyTimeout: Delay to wait (in seconds) for a device to reply with a REPORT on a GET command
- DeviceRelaxDelay: Delay between two subsequent packets sent to one device, measured in ticks (10 ms). Some slow devices might need about 10 to respond correctly to burst of packets
- SerialAPITimeout: Extra time to be added to Serial API timouts. Set up to 1.0-3.0 sec in case of slow channel toward Z-Wave chip (e.g. in cloud applications)
- Command Class-Specific Settings
  - Wakeup -> WakeupInterval: Default Wakeup Interval
  - Scene Actuator Conf -> Max Scenes: Maximum number of Scenes supported
  - Scene Controller Conf -> Max Scenes: Maximum number of Scenes supported
  - Protection -> Mode: Default Protection Mode
  - SensorMultilevel -> Fahrenheit: Flag what temperature scale is used
  - SwitchAll -> Mode: Default Switch All mode
  - MultiCmd -> MaxNum: The maximum number of commands within multi-command encapsulation. The optimal value would be even, but there are many broken devices in the market that do not support this. A lower number means less efficient but more robust against faulty devices.
  - Firmware Update -> Fragment Size: Fragment size on 3rd gen RaZberry and 3rd gen UZB cannot be more than 32 (max packet size was 37, with possible CRC it gives 32). On UZB and new 5gen it can be up to 40 bytes
  - ThermostatSetPoint: -> Fahrenheit: Flag what temperature scale is used

- Controller: Description of how Z-Way will behave as a device in the network. This entry has the following subentries:
  - NodeInformationFrame: The command class Z-Way is described as "supported" in the network
  - SecureNodeInformationFrame: Command Classes available in secure environment
  - InstanceNodeInformationFrame: Command Classes in Instances if Multi Channel is emulated
  - VersionID =iD: Versions to be reported in Command Class Version Get Command for all Command Classes announced in NIF
  - Name: Default Node Name reported by NodeNaming Report
  - Location Default Node Location of Controller reported by NodeNaming Report
  - AppVersion: Application Version reported by ManufacturerSpecific Report
  - Manufacturer Specific: Values report by ManufacturerSpecific Report
  - SpecificDeviceClass: Specific Device Class reported
  - GenericDeviceClass: Generic Device Class reported
  - Icons: Z-Wave Plus Icons
  - Lifeline: Defines how many devices will be in Lifeline Association Group
  - CommandClassSupportedVersion: Defines the version numbers of the supported command classes
  - Channels: Defines the simulated channels

The file **Profiles.xml** contains the EnOcean profile definition. This clearly reflects the official profile definitions published by the EnOcean alliance.

The file **Rules.xml** is a legacy file.

translations: XMLs mapping Ids to text

```
translations
├─ AEC.xml:  Advanced Energy Framework
├─ Alarms.xml:  Alarm conditions
├─ BarrierSignals.xml
├─ ColorCapabilities.xml
├─ DeviceClasses.xml:  Z-Wave Classic Device Classes
├─ LocEvents.xml:  Door Lock Events
├─ Scales.xml:  Sensor Multilevel and Meter Scales
├─ SDKIds.xml:  Major Minor into SDK versions
├─ ThermostatModes.xml
├─ VendorIds.xml:  Vendor Id into Vendor Name
└─ ZWavePlus.xml:  Z-Wave Plus Role Types and Network Types
```

All files in this subfolder are XML files and some of them require local language translations, as described in Chapter 10.3.

ZDDX: Device Description Files

ZDDX files (Z-Wave Device Description XML Files) are XML files containing verbal description of a specific Z-Wave device that cannot be called from the device itself during the interview process: They are:

- The naming of Association Groups. Modern Z-Wave device provides them in English language using the Association Group Information Command Class. The ZDDX file provides this information for older devices and in various languages.
- Configuration Parameters and Values: It is always possible to set a configuration value knowing the integer values from the device manual. Z-Way offers a convenient way to set Z-Wave configuration values utilizing the information from ZDDX files.
- An Image of the device.
- Information on inclusion, exclusion and wakeup processes.

It is possible to add your own ZDDX files, but Z-Way uses an index file ZDDX.indx to access them. Once a new file is added, run **python MakeIndex.py**.
Chapter 13.3 explain how to add and to submit new own ZDDX files and how to extend them.

Htdocs: The web servers document folder

```
htdocs :  web server doc folder
 ├─ config:  link from /config into web space
 ├─ ZDDX: link from /ZDDX into web space
 ├─ expert:  Z-Wave Expert User Interface
 ├─ smarthome:  Smart Home User Interface
 ├─ translation
 └─ index.html
```

This subfolder is the root folder of the embedded webserver. The index.html redirects to smarthome/index.html.

automation: The JavaScript sub system

```
automation
 ├─ classes:  Some base classes of the JS system
 ├─ core:  The core classes of the JS system
 ├─ defaultConfigs:  A collection of default config.json files
 ├─ lang:  Translations of JS system messages
 ├─ lib:  Some utility scripts of the JS system
 ├─ modules:  The preinstalled apps
 ├─ userModules:  Downloaded apps
 │   └─ uploadModule.sh:  Utility to upload own created apps
 └─ storage:  The central folder to store all user data and settings
     ├─ configjson-XXXXXXX: Configuration Settings of Smart Home UI
     ├─ dedevicesjson-XXXXXXX: complete device description in German)
     ├─ endevicesjson-XXXXXXX: complete device description in English
     ├─ expertconfigjson-XXXXXXX: Configuration Settings of Expert UI
     ├─ history-XXXXXXX: Data for 24 hour history display
     └─ ...:  more json files containing data
```

The subfolder for automation contains the whole JavaScript (JS) subsystem including the

- source code of the software core (classes, core, lib),
- the storage for the apps (modules, user modules),
- the central storage for configs, images, uploads, history, logging, etc.,
- some default files for factory default reset and recovery from failures.

The most sensitive file is configjson-XXX since it contains the information about the user accounts including login name, password, and recovery email.
For more information about the creation of new modules that will be placed into the user modules, please refer to Chapter 13.2.

11.4.2 The use of the C-Library

The Z-Way library is a middleware between Silicon Labs Z-Wave transceiver and your application. Z-Way offers pretty high level API to control Z-Wave devices and manage wireless network.
Interaction with the library covers three aspects:

- sending commands to Z-Wave devices;
- sending network management commands to the transceiver;
- receiving updates from the network.

Sending commands

Every command request generates an outgoing packet (job). Before generating a packet, library will validate parameters and check whether the command is supported by the recipient. In case of failure command will return error

immediately.
Once a job is generated, it is placed into outgoing queue for future send. The queued jobs are handled internally by Z-Way engine according to commands priorities, nodes states and capabilities, transceiver state etc.
Once the job is sent, it must be first confirmed it was successfully delivered to the Z-Wave stack, and then confirmed it was delivered to the recipient. All these operations are performed asynchronously, so command may provide a callback function to call in case of success or failure if it is needed to know delivery result.
After the delivery was confirmed, command is considered executed. If it was a state request command (i.e. `SensorMultilevel Get`), response packet may be delayed (or even not sent at all), so command's success/failure callbacks cannot be used to get requested state immediately.

Receiving updates

All incoming packets from the Z-Wave network are automatically parsed by Z-Way and stored in special variables called data holders. Data holder is a named variable that stores a value along with its data type and time the value was last updated and "invalidated". Each data holder may also contain a set of child data holders, so they form a hierarchical data storage. Data holders also support callbacks, so custom code may be executed each time the value is updated.
For example, `level` data holder stores dimming level of a dimmer. Once application executes a `Get` command for that dimmer, Z-Way will update `invalidateTime` property on the `level` data holder, so application knows the current value is assumed to be outdated, but the new one was not received yet.
Once Z-Way received a packet with new value of the dimmer, it will store it in `level data holder` and update `updateTime` property. Since `updateTime` is greater that `invalidateTime`, the value is considered valid now.
Z-Wave device can also send unsolicited state reports to controller (without a request from controller's side; e.g. due to local operation or periodically). Due to asynchronous nature of Z-Wave protocol, controller can't tell whether the packet was sent unsolicited or it is a response to the previous command. So unsolicited packet will be handled the same way exactly.

Command Classes

Z-Way inherits structure of Z-Wave protocol and divides data holders and commanda on different Command Classes (CC). Command Classes are building blocks of Z-Wave functionality. For example, dimming is provided by Command Class `SwitchMultilevel`, relay operation by Command Class `SwitchBinary`, sensors by command class `SensorMultilevel` and `SensorBinary` etc. Please consult Z-Wave protocol basics to understand Z-Wave Command Classes.
All Command Classes share a minimal subset of common data holders:

- `supported` says if CC is supported by the device (it implements that functionality) or only controlled (it can control other devices implementing that functionality).
- `version` stores version of the CC. Used internally to know how to deal with that Command Class.
- `security` tells if CC communications should be encrypted using Z-Wave AES security mechanism.
- `interviewDone` and `interviewCounter` describe the status of initial interview process during which Z-Way asks the device about its CC capabilities. If the interview is incomplete, Z-Way might fail to use some Command Classes with this device. All Z-Wave certified devices MUST pass interview process.

All the other data holders are specific to each Command Class. For example, `SwitchMultilevel Command Class` contains `level` data holder, `SensorBinary` has two-level storage, grouping data by sensor types: `0 -> sensorTypeString, level, 5 -> sensorTypeString, level, ...` where type identifiers are Z-Wave specific constants. Every Z-Wave specific constant value will have corresponding verbal description (in case of `SensorBinary` it is in `sensorTypeString` data holder).
Some Command Classes are hidden under the hood of Z-Way: `MultiCmd, Security, CRC16, MultiChannel, ApplicationStatus`. They're handled internally by Z-Way software, and shouldn't be used directly.
Some Command Classes have no public APIs, but their data holders may be very useful in your application: `Association-GroupInformation, DeviceResetLocally, ManufacturerSpecific, Version, ZWavePlusInfo`.
All the remaing Command Classes have their `Get` and `Set` commands specific to functionality of the Command Class. Consult `CommandClassesPublic.h` header file for more info about available commands for different Command Classes and their meaning.

Network management

Z-Way offers API for network management operations: include new devices, exclude devices, discover neighbor devices, remove failed nodes, frequency selection, controller reset etc. These functions are described in `ZWayLib.h` header file.

Z-Way also provides a low level access to Z-Wave transceiver functionality through Silicon Labs Serial API. These functions are provided by Function Classes. You should use them only if you have deep knowledge of Z-Wave networking. Check `FunctionClassesPublic.h` for more info.

Using Z-Way Library in C-Code

To use Z-Way one need to include few header files:

Listing 2.

```
#include <ZWayLib.h>
\#include <ZLogging.h>
```

Z-Way will need to know where to write the log to, so first of all you need to create logging context using `zlog_create()` call. You can disable logging by passing NULL instead of logging context to Z-Way.
Then create new a Z-Way context using `zway_init()`. It will only allocate memory, open log files, initialize internal structures. At this point you can already attach your handlers on new `device/instance/Command Class` creation (you will also be able to do it at any time later). Do it using `zway_device_add_callback()` call.
Warning: you should initialize ZWay pointer with NULL before passing it to `zway_init`!
Executing `zzway_start()` will open serial port and start a new thread that will handle all communications with the transceiver. From now Z-Way can receive packets from the network, but can not parse them yet, since devices were not discovered yet. All received packets will just be queued to be parsed later after discovery process.
Last step to run Z-Way is `zway_discover()` call. It will start communications with the Z-Wave transceiver and ask about devices in the network, their capabilities, network state etc. During discovery phase Z-Way will create structures for all devices and load saved data from file stored in `config/zddx` folder.
From now on, Z-Way is ready to operate. Incoming events will trigger callback functions attached by application, and executing commands will put new packets in the queue.
You will also need few other functions `zway_is_running(), zway_is_idle(), zway_stop(), zway_terminate()` to handler termination process.
The link

`https://storage.z-wave.me/fileadmin/z-way-test.tgz`

downloads a very simple test project using the Z-Way core library `libzway.so`. The project contains a simple `mains.c` plus a `Makefile`. A Z-Way installation is required too. To compile the test project, some requirements need to be met:

- GNU-based tool chain with a compiler, linker, etc.
- Copy main.c and Makefile into the root folder of Z-Way
- Check the path to library and header files and adapt the Makefile is needed
- Make sure the following libraries are installed:
  - libxml2 (apt-get install libxml2-dev)
  - libarchive (apt-get install libarchive-dev)
  - libcrypto (apt-get install libssl-dev)

Executing the Makefile will generate a binary executable **z-way-test**
in the same folder. If Z-Way was just downloaded the code will assume a virtual serial Z-Wave device on **/dev/ttyUSB0**
if your virtual device is a different node, just make a symlink like **ln -s /dev/ttyACM0 /dev/ttyUSB0**
or change `main.c` file.
Now start the code from the folder with **LD_LIBRARY_PATH=./libs ./z-way-test**
The header file `Z-Waylib.h` in `libzway` gives a brief explanation of the calls into the library. The demo file `main.c` demonstrates the use of the calls. Figure 11.8 shows the small help page of the test software.

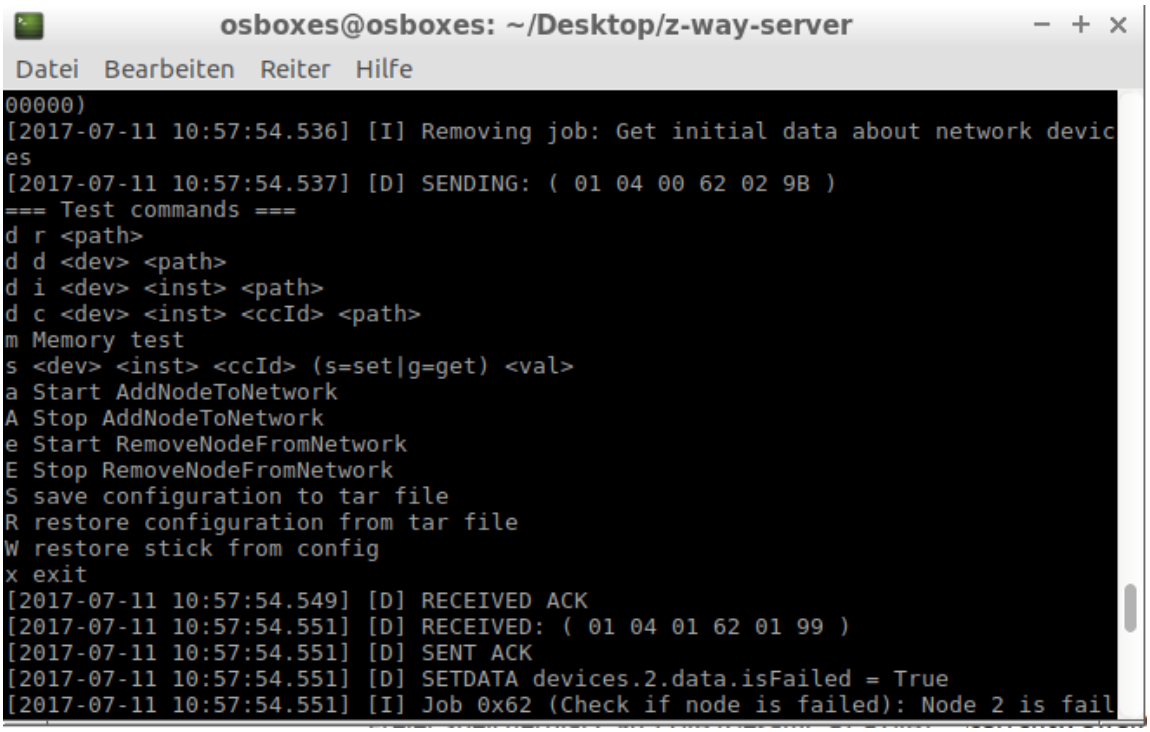


Figure 11.8: Terminal running z-way-test

12 The JavaScript Engine

The Z-Way core function engine is the so called JavaScript (JS) automation system. It uses the APIs of the technology-dependent 'drivers' and delivers all the functions and interface for running a Smart Home controller:

- It **unifies the functions and properties of the physical devices** to a common device structure, the virtual Device (vDev).
- It allows to **create own virtual devices** not related to physical devices.
- It allows to **dynamically load 'plugin' modules** - also written in Javascript or even in native C - that extend the function of the JS core and deliver furter functionality. Users see these modules as apps. For more information about apps please refer to chapter 6.
- It organizes the **communication between the virtual devices** as event bus. Every vDev can inject events into this bus and every vDev can read events from this bus.
- It provides a **higher layer API** that is is used among others by the Smart Home User Interface.

This chapter explains the different building blocks of the JS Engine:

- Javascript Interpreter/Compiler Core Engine
- Virtual Devices
- Event Bus
- User Interface elements referring to the different virtual devices
- The structure of the Apps

12.1 JavaScript API

Z-Way uses the JavaScript engine provided by Google referred to as V8. You find more information about this JavaScript implementation on

`https://code.google.com/p/v8/.`

V8 implements JavaScript according to the specification ECMA 5 [1].

Please note that this V8 core engine only implements the very basic JS functions and need to be extended to be usable in a Smart Home environment.

Z-Way extends the basic functionality provided by V8 with plenty of application-specific functions.

Javascript code can be executed on the server side and certain functions of the JS core are available on the client side as well since most modern web browsers have a built-in Javascript engine as well. The bridge between the server side JS and the web browser client is a built-in web server. This is the same embedded web server serving all the web browsers HTML pages etc.

There are three ways to run JavaScript code in Z-Way backend.

- It can be executed in via web browser using a special URL string containitg the code to be executed (using /JS/Run/ prefix),
- It can be stored as a file on the server in automation folder and executed using 'executeFile' function,
- It can be implemented as an App running.

All options have their pros and cons. Running JS code via the browser is a very nice and convenient way to test things but the code is not persistent across Z-Way restarts.

Storing the it in a file allows to run it on Z-Way start (if 'executeFile("myfile.js")' is placed in main.js) but is not really convenient to distribute.

Writing a module requires more knowledge, but includes a nice graphical interface for App configuration. Upload your App in the Z-Way App Store for easy deployment and distribution of your App.

Check Z-Way App Store on

`https://developer.z-wave.me/`

for more information. There are many other open source Apps made by the community.

Accessing a server side JS function from the web browsers client side is easy. Just call

`http://YOURIP:8083/JS/Run/<anyJScode>`

Please note that all accesses using the embedded webserver require an authentication of the web browsers instance. Please refer to chapter 13.1 for details how to authenticate in the Z-Way web server.

[1] http://www.ecma-international.org/publications/standards/Ecma-262.htm

Z-Way offers one central object with the name 'zway' . This object encapsulates all the Z-Wave variables and functions known from the Z-Wave Device API describes in chapter 11.3.
Hence its possible to use the very same functions of the Z-Wave Device API using the JS engine. The zway objects internal structure is shown in figure 11.2 and the data elements are describes in Annex 11.3.1.
The functions can be accessed using the web browsers function like

```
http://YOURIP:8083/JS/Run/zway.devices[x].*
```

Due to the scripting nature of JavaScript it is possible to 'inject' code at run time using the interface. Here a nice example how to use the Java Script setInterval function:

Listing 1. Polling device #2

```
/JS/Run/setInterval(function() {
zway.devices[2].Basic.Get();
}, 300*1000);
```

This code will, once 'executed' as a URL within a web browser, calls the Get() command of the command class Basic of Node ID 2 every 300 seconds.
A very powerful function of the JS API is the ability to bind functions to certain values of the device tree. they get then executed when the value changes. Here an example for this binding. The device No. 3 has a command class SensorMultilevel that offers the variable level. The following call - both available on the client side and on the server side - will bind a simple alert function to the change of the variable.

Listing 2. Bind a function

```
zway.devices[3].SensorMultilevel.data[1].val.bind(function() {
debugPrint('CHANGED TO: ' + this.value + '\n');
});
```

12.2 Z-Way extensions to the JavaScript Core

Z-Way provides some extensions to the JS core that are not part of the ECMA functionality mentioned above.

12.2.1 HTTP Access

The JavaScript implementation of Z-Way allows directly accessing HTTP objects.
The http request is much like jQuery.ajax(): r = http.request(options);
Here's the list of options:

- url - required. Url you want to request (might be http, https, or maybe even ftp);
- method – optional. HTTP method to use (currently one of GET, POST, HEAD). If not specified, GET is used;
- headers – optional. Object containing additional headers to pass to server:

  Listing 1.

  ```
  headers: {
      "Content-Type": "text/xml",
      "X-Requested-With": "RaZberry/1.5.0"
  }
  ```

- data – used only for POST requests. Data to post to the server. May be either a string (to post raw data) or an object with keys and values (will be serialized as 'key1=value1&key2=value2&…');
- auth – optional. Provides credentials for basic authentication. It is an object containing login and password:

  Listing 2.

  ```
  auth: {
      login: 'username',
      password: 'secret'
  }
  ```

- contentType – optional. Allows overriding content type returned by the server for parsing data (see below);

- async – optional. Specifies whether request should be sent asynchronously. Default is false. In case of synchronous request, result is returned immediately (as function return value); otherwise, function exits immediately, and response is delivered later thru callbacks.
- success, error and complete – optional, valid only for async requests. Success callback is called after successful request, error is called on failure, complete is called nevertheless (even if success/error callback produces exception, so it is like the 'finally' statement);

Response (as stated above) is delivered either as function return value, or as callback parameter. It is always an object containing the following members:

- status – HTTP status code (or -1 if some non-HTTP error occurred). Status codes from 200 to 299 are considered success;
- statusText – status string;
- URL – response URL (might differ from URL requested in case of server redirects);
- headers – object containing all the headers returned by server;
- contentType – content type returned by server;
- data – response data.

Response data is handled differently depending on content type (if contentType on request is set, it takes priority over server content type):

- application/json and text/x-json are returned as JSON object;
- application/xml and text/xml are returned as XML object;
- application/octet-stream is returned as binary ArrayBuffer;
- string is returned otherwise.

In case data cannot be parsed as valid JSON/XML, it is still returned as string, and additional parseError member is present.

Listing 3.

```
http.request({
url: "http://server.com" (string, required),
method: "GET" (GET/POST/HEAD, optional, default "GET"),

headers: (object, optional)
{
"name": "value",
...
},

auth: (object, optional)
{
"login": "xxx" (string, required),
"password": "***" (string, required)
},

data: (object, optional, for POST only)
{
"name": "value",
...
}
-- OR --
data: "name=value&..." (string, optional, for POST only),

async: true (boolean, optional, default false),

timeout: (number, optional, default 20000)

success: function(rsp) {} (function, optional, for async only),
error: function(rsp) {} (function, optional, for async only),
complete: function(rsp) {} (function, optional, for async only)
});


response:
```

```
{
status: 200 (integer, -1 for non-http errors),
statusText: "OK" (string),
url: "http://server.com" (string),
contentType: "text/html" (string),
headers: (object)
{
"name": "value"
},
data: result (object or string, depending on content type)
}
```

12.2.2 XML parser

ZXmlDocument object allows converting any valid XML document into a JSON object and vice versa.

var x = new ZXmlDocument()

Create new empty XML document

x = new ZXmlDocument("xml content")

Create new XML document from a string

x.root

Get/set document root element. Elements are got/set in form of JS objects:

Listing 4.

```
{
    name: "node_name", - mandatory
    text: "value", - optional, for text nodes
    attributes: { - optional
     name: "value",
     ...
    },
    children: [ - optional, should contain a valid object of same type
     { ... }
    ]
}
```

For example:

Listing 5.

```
(new ZXmlDocument('<weather><city id="1"><name>Zwickau</name>
<temp>2.6</temp></city>
<city id="2"><name>Moscow</name><temp>-23.4</temp></city>
</weather>')).root =
{
   "children":[
      {
         "children":[
            {
               "text":"Zwickau",
               "name":"name"
            },
            {
               "text":"2.6",
               "name":"temp"
```

```
            }
         ],
         "attributes":{
            "id":"1"
         },
         "name":"city"
      },
      {
         "children":[
            {
               "text":"Moscow",
               "name":"name"
            },
            {
               "text":"-23.4",
               "name":"temp"
            }
         ],
         "attributes":{
            "id":"2"
         },
         "name":"city"
      }
   ],
   "name":"weather"
}
```

x.isXML

This hidden read-only property allows detecting if the object is an XML object or not (it is always true).

x.toString()

Converts XML object into a string with valid XML content.

x.findOne(XPathString)

Returns first matching to XPathString element or null if not found.

Listing 6.

```
x.findOne('/weather/city[@id="2"]') // returns only city tag for Moscow
x.findOne('/weather/city[name="Moscow"]/temp/text()') // returns temperature in Moscow
```

x.findAll(XPathString)

Returns array of all matching to XPathString elements or empty array if not found.

Listing 7.

```
x.findAll('/weather/city') // returns all city tags
x.findAll('/weather/city/name/text()') // returns all city names
```

x.document

A hidden property that refers to the document root.

XML elements

Each XML element (tag) in addition to properties described above (text, attributes, children) have hidden read-only property parent pointing to parent object and the following methods:

- insertChild(element) Insert new child eleemnt
- removeChild(element) Remove child element
- findOne(XPathString) Same as on root object, but relative (no leading / needed in XPathString
- findAll(XPathString) Same as on root object, but relative (no leading / needed in XPathString

ZXmlDocument is returned from http.request() when content type is 'application/xml', 'text/xml' or any other ending with '+xml'. Namespaces are not yet supported.

12.2.3 Cryptographic functions

crypto object provides access to some popular cryptographic functions such as SHA1, SHA256, SHA512, MD5, HMAC, and provides good random numbers.

var guid = crypto.guid()

Provides standard GUID in string format.

var rnd = crypto.random(n)

Generates n random bytes. Returned value is of type ArrayBuffer. To convert it into array, use this trick:

Listing 8.

```
rnd = (new Uint8Array(crypto.random(10)));
```

var dgst = crypto.digest(hash, data, ...)

Returns digest calculated using selected hash algorithm. It supports virtually all the algorithms available in OpenSSL (md4, md5, mdc2, sha, sha1, sha224, sha256, sha384, sha512, ripemd160). If no data parameters specified, it returns a digest of an empty value. If more than one data parameters are specified, they're all used to calculate the result. Data parameters may be of several types: strings, arrays, ArrayBuffers. Return value is of type ArrayBuffer.
There are also a few shortcut functions for popular algorithms: 'md5', 'sha1', 'sha256', 'sha512'. For example, these calls are equivalent:

Listing 9.

```
dgst = crypto.digest('sha256', data);
dgst = crypto.sha256(data);
```

var hmac = crypto.hmac(cipher, key, data, ...)

Returns hmac calculated using selected hash algorithm. Hash algorithms are the same as for digest() function.
Key parameter is required.
If no data parameters specified, it returns a HMAC of an empty value. If more than one data parameter is specified, they're all used to calculate the result. Key and data parameters may be of different types (strings, arrays, ArrayBuffers). Return value is of type ArrayBuffer.
There are also a few shortcut functions for popular algorithms: 'hmac256', 'hmac512'. For example, these calls are equivalent:

Listing 10.

```
dgst = crypto.hmac('sha256', key, data);
  dgst = crypto.hmac256(key, data);
```

12.2.4 Sockets functions

Socket module allows easy access to TCP and UDP sockets from JavaScript. Both connection to distant ports and listening on local are available. This API fully mirrors into JavaScript POSIX TCP/IP sockets. This can be used to control third party devices like Global Cache or Sonos as well as emulating third party services.
To start communication, one needs to create socket and either **connect** it or **listen** it. **onrecv** method is called on data receive from remote, while **send** is used to send data to remote side.
The example below dumps to log file response to http://ya.ru:80/ (raw HTTP protocol is used as an example).

Listing 11.

```
var sock = new sockets.tcp();

sock.onrecv = function(data) {
    debugPrint(data.byteLength);
};

sock.connect('ya.ru', 80);

sock.send("GET / HTTP/1.0\r\n\r\n");
```

Here is an example of TCP echo server on port 8888:

Listing 12.

```
var sock = new sockets.tcp();

sock.bind(8888);

sock.onrecv = function(data) {
    this.send(data);
};

sock.listen();
```

And echo server for UDP:

Listing 13.

```
var sock = new sockets.udp();

sock.bind(8888);

sock.onrecv = function(data, host, port) {
    this.sendto(data, host, port);
};

sock.listen();
```

Important! Callbacks can only be specified before the connection is established.
“this“ inside callbacks refers to the socket object itself.
Detailed description of Socket API:

- bind(ip, port) or bind(port) binds socket to port (integer number). ip should be a string like "192.168.0.1". If omited "0.0.0.0" is used (bind on all IP addresses of all interfaces). Returns false on error.
- connect(ip, port) connects to remote side ip:port. TCP sockets requires this call before sending data. For UDP sockets it is optional, but once used, it allows using send call instead of sendto call. Returns false on error.
- listen() starts listening port (this is required not only for TCP, but for UDP too). Returns false on error.
- close() initiate close of socket.
- send(data) sends data to connected or accepted socket.
- sendto(data, host, port) sends data to a non-connected UDP socket.
- onrecv(data, host, port) called on new data receiption from remote side. For UDP sockets and connected TCP sockets "this" object refers to the socket itself, while for accepted TCP sockets "this" reffers to the client’s individual objects.

- onconnect(remoteHost, remotePort, localHost, localPort) is called only for TCP sockets on new connection accept or for established connection on client socket. "this" refers to the client individual socket object.
- onclose(remoteHost, remotePort, localHost, localPort) called on socket close by remote or due to close() call. Note that for TCP sockets this callback is called for client sockets on connection close and for bindound listening socket if close() is called. "this" object will be defined like in onrecv.
- reusable() sets SO_REUSEADDR socket option to allow multiple bind() on the same port.
- broadcast() sets SO_BROADCAST socket option to allow sending broadcast UDP messages.
- multicastAddMembership(multicastGroup) subscribe socket to multicast group
- multicastDropMembership(multicastGroup) unsubscribe socket from multicast group

12.2.5 WebSockets functions

Socket module also implements WebSockets (RFC 6455). WebSocket API is made to be compatible with browser implementations (some rarely used functions are not implemented, see below).
The example below implements basic application using the WebSockets client:

Listing 14.

```
var sock = new sockets.websocket("ws://echo.websocket.org");

sock.onopen = function () {
    debugPrint('connected, sending ping');
    sock.send('ping');
}

sock.onmessage = function(ev) {
    debugPrint('recv', ev.data);
}

sock.onclose = function() {
    debugPrint('closed');
}

sock.onerror = function(ev) {
    debugPrint('error', ev.data);
}
```

Next example shows basic application using WebSockets server:

Listing 15.

```
var sock = new sockets.websocket(9009);

sock.onconnect = function () {
    debugPrint('client connected, sending ping');
}

sock.onmessage = function(ev) {
    debugPrint('recv', ev.data);
    sock.send('pong');
}

sock.onclose = function() {
    if (this === sock) {
        debugPrint('server websocket closed');
    } else {
        debugPrint('client disconnected');
    }
}

sock.onerror = function(ev) {
    debugPrint('error', ev.data);
```

```
}
```

Detailed description of WebSocket API:

- sockets.websocket(url, [protocol], [ssl_ca_filepath], [ssl_cert_filepath, ssl_private_key_filepath]) creates new client WebSocket and connects to the specified URL (should be a string like "ws://host:port" or "wss://host:port" for SSL channel). Optional protocol parameter can be used to specify protocol from server capabilities (comma separated string), default is "default". To use a specific CA file instead of system default use ssl_ca_filepath. To send client certificate use ssl_cert_filepath and ssl_private_key_filepath.
- sockets.websocket(port) creates new WebSocket server on port.
- close() initiate close of WebSocket.
- send(data) sends data to WebSocket. data can be array, ArrayBuffer (sent as binary) or string (sent as text).
- onmessage(event) called on new data reception from remote side. Object event contains only data property. Other properties mentioned in RFC 6455 are not supported.
- onopen() or onconnect() called on connection establish. Compared RFC 6455 event parameter is not passed.
- onclose() called on WebSocket close by remote or due to close() call. For server side will be called on client instance and on listening instance (use this to differenciate). Note that if close() is called before connection was established, onclose() is not executed. Compared RFC 6455 event parameter is not passed.
- onerror(event) called on error. For example, host or port unreachable. Note that new sockets.websocket(..) can throw an exception on DNS resolution error or on network unreachable. Other errors will be reported via onerror. Parameter event contains only propery data. Other properties from RFC 6455 are not implemented.

12.2.6 MQTT functions

MQTT module allows to connect to MQTT broker from JavaScript. Both subscribtion to remote to topics and publishing own topics are possible. TCP and WebSocket transports are supported as well as TLS (requires libmosquitto 2.1.0 or upper).
The example below connects to a server using and publishes a topic.

Listing 16.

```
var m = new mqtt("broker.emqx.io", 1883);

m.onconnect = function() {
debugPrint("Connected");
};

m.ondisconnect = function() {
debugPrint("Disconnected");
};

m.onpublish = function() {
debugPrint("Published");
};
it will work when the sent message is published;

m.onsubscribe = function () {
debugPrint("Subscribed");
};

m.onmessage = function (topic, message) {
debugPrint("New topic " + topic + ": " + message);
};

m.connect();
```

Here is an example of WebSocket based connection:

Listing 17.

```
var m = new mqtt("ws://broker.emqx.io", 8083);
```

Important! Callbacks can only be specified before the connection is established.
“this“ inside callbacks refers to the mqtt object itself.
Detailed description of WebSocket API:

- new mqtt(host, port, [login, password], [name]) host can be just a host name or ws://hostname or wss://hostname. login/password allows to use authentication if the broker requires it. name is for logging only.
- onconnect() called when the connection is established
- ondisconnect() called when the client is disconnected
- onpublish() called when the message is successfully published
- onsubscribe() called when the broker confirms the subscription
- onmessage(topic, message) called on a new publication received

You can to set the TSL settings before the connection is established. Configure the client for certificate based SSL/TLS support:

- tlsset(ca_file, ca_path, cert_file, key_file) set TLS settings. ca_file: path to a file containing the PEM encoded trusted CA certificate files. Either cafile or ca_path must not be empty string. ca_path: path to a directory containing the PEM encoded trusted CA certificate files. Either cafile or ca_path must not be empty string. cert_file: path to a file containing the PEM encoded certificate file for this client. If empty string, key_file must also be empty string and no client certificate will be used. key_file: path to a file containing the PEM encoded private key for this client. If empty string, cert_file must also be empty string and no client certificate will be used.
- tlsinsecure() disables the verification of the server hostname in the server certificate. This is unsecure, use for debugging only!
- tlsadvanced(cert_reqs, tls_version, ciphers) Set advanced SSL/TLS options. cert_reqs: 1 to enable the server certificate verification, or 0 to disable during debugging. tls_version: tlv1, tlsv1.1, tlv1.2 or empty for default. ciphers: a string describing the ciphers available for use. If empty string, the default ciphers will be used.

The MQTT object provides the following methods:

- connect() to connect to the broker
- publish(topic, message) to pblish a message on a given topic
- subscribe(topic) to subscribe to a topic
- unsubscribe(topic) to unsubscribe to a topic
- disconnect() to disconnect from the broker, you can connect again later

12.2.7 Other JavaScript Extensions

fs.list(folder)

This returns the list of items in the folder or undefined if the folder does not exist.

fs.stat(file)

This returns one of the following values:

- 1) undefined if object does not exist or not readable
- 2) object { type: ’file’, size: <size>} if it is a file
- 3) object { type: ’dir’ } if it is a folder

fs.loadJSON(filename)

This function reads a file from the file system and loads it into the memory. The file must contain a valid JSON object. The only argument is the name of the file including relative pathname to the automation folder. Returns the full JSON object or null in case of error.

fs.load(filename)

This function reads a file from the file system and returns its content as a string. The only argument is the name of the file including relative pathname to the automation folder. Returns null in case of error.

executeFile(filename) and executeJS(string)

Loads and executes a particular JavaScript file from the local filesystem or executes JavaScript code represented in string (like eval in browsers).
The script is executed within the global namespace.

Remark: If an error occurrs during the execution, it won't stop from further execution, but erroneous scripts will not be executed completely. It will stop at the first error. Exceptions in the executed code can be trapped in the caller using standard try-catch mechanism.

system(command)

The command system() allows executing any shell level command available on the operating system. It will return the shell output of the command. By default the execution of system commands is forbidden. Each command executed need to be permitted by putting one line with the starting commands in the file automation/.syscommands or in an different automation folder as specified in config.xml.

Timers

Timers are implemented exactly as they are used in browsers. They are very helpful for periodical and delayed operations. Timeout/period is defined in milliseconds.

- timerId = setTimeout(function() , timeout)
- timerId = setInterval(function() , period)
- clearTimeout(timerId)
- clearInterval(timerId)

loadObject(object_name) and saveObject(object_name, object)

Loads and saves JSON object from/to storage. These functions implement flat storage for application with access to the object by its name. No folders are available.
Data is saved in automation/storage folder. Filenames are made from object names by stripping characters but [a-ZA-Z0-9] and adding checksum from original name (to avoid name conflicts).

exit()

Stops JavaScript engine and shuts down Z-Way server

allowExternalAccess(handlerName) and listExternalAccess()

allowExternalAccess allows registering HTTP handler. handlerName can contain strings like aaa.bbb.ccc.ddd — in that case any HTTP request starting by /aaa/bbb/ccc/ddd will be handled by a function aaa.bbb.ccc.ddd() if present, otherwise aaa.bbb.ccc(), ... up to aaa(). The handler should return object with at least property status and body (one can also specify headers like it was in http.request module).
listExternalAccess returns array with names of all registered HTTP handlers.
Here is an example how to attach handlers for /what/timeisit and /what:

Listing 18.

```
what = function() {
  return { status: 500, body: 'What do you want to know' };
};

what.timeisit = function() {
  return { status: 200, body: (new Date()).toString() }
};

allowExternalAccess("what");
allowExternalAccess("what.timeisit");
```

debugPrint(object, object, ...)

Prints arguments converted to string to Z-Way console. Very useful for debugging. For convenience, one can map 'console.log()' to debugPrint().
This is how it was done in automation/main.js in Z-Way Home Automation engine:

Listing 19.

```
var console = {
    log: debugPrint,
    warn: debugPrint,
    error: debugPrint,
    debug: debugPrint,
    logJS: function() {
        var arr = [];
        for (var key in arguments)
            arr.push(JSON.stringify(arguments[key]));
        debugPrint(arr);
    }
};
```

12.2.8 Debugging JavaScript code

Change in config.xml debug-port to 8183 (or some other) turn on V8 debugger capability on Z-Way start.

Listing 20.

```
<config>
    ...
    <debug-port>8183</debug-port>
    ....
</config>
```

node-inspector debugger tool is required. It provides web-based UI for debugging similar to Google Chrome debug console.
You might want to run debugger tool on another machine (for example if it is not possible to install it on the same box as Z-Way is running on).
Use the following command to forward debugger port defined in config.xml to your local machine:

```
ssh -N USER@IP_OF_Z-WAY_MACHINE -L 8183:127.0.0.1:8183
```

(for RaZberry USER is pi)
Install node-inspector debugger tool and run it:

```
npm install -g node-inspector node-inspector –debug-port 8183
```

Then you can connect to

```
http://IP_OF_MACHINE_WITH_NODE_INSPECTOR:8080/debug?port=8183
```

If debugging is turned on, Z-Way gives you five seconds during startup to reconnect debugger to Z-Way (refresh the page of debugger Web UI within these five seconds). This allows you to debug startup code of Z-Way JavaScript engine from the very first line of code.

12.3 The virtual device concept (vDev)

A virtual device is a data object within the JS engine. Virtual devices have properties and functions. Most virtual devices represent a physical device or a part of a physical device but virtual devices are not limited to this. Virtual devices can be pure dummy device doing nothing but pretenting to be a device (There is an app called **'Dummy Device'** that works exactly like this). Virtual devices can also connect to services via TCP/IP.
The purpose of virtual devices is to unify the appearance on a graphical user interface and to unify the communication between them. At the level of virtual devices and EnOcean controller can switch a Z-Wave switch and trigger a rule in a cloud service.

12.3.1 Names and Ids

Every virtual device is identified by a simple string type id. For all virtual devices that are related to physical Z-Wave devices the device name is auto-generated by the module (app) 'Z-Wave' following this logic:

ZWayVDev_[Node ID]:[Instance ID]:[Command Class ID]:[Scale ID] The Node Id is the node id of the physical device, the Instance ID is the instance id of the device or '0' if there is only one instance. The command class

deviceType	Metrics	Commands	Examples
battery	probeTitle,scaleTitle, level, icon, title	-	-
doorlock	level, icon, title	open or close	apiURL/devices/:deviceId/command/open
thermostat	scaleTitle, min, max, level, icon, title	exact with get-param level	apiURL/devices/:deviceId/command/exact? level=22.5
switchBinary (Thermostat)	level, icon, title	on, off or update	apiURL/devices/:deviceId/command/on
switchBinary	level, icon, title	on, off or update	apiURL/devices/:deviceId/command/on
switchMultilevel	level, icon, title	on Set(255), off Set(0), min Set(10), max Set(99), increase Set(l+10), decrease Set(l-10), update, exact + get params level	apiURL/devices/:deviceId/command/exact? level=40
switchMultilevel (Blinds)	level, icon, title	up Set(255), down Set(0), upMax Set(99), increase Set(l+10), decrease Set(l-10), startUp StartLevelChange(0), startDown StartLevelChange(1), stop StopLevelChange(), update, excactSmooth + get params level	apiURL/devices/:deviceId/command/stop
sensorBinary	probeTitle, level, icon, title	update	apiURL/devices/:deviceId/command/update
sensorMultilevel	probeTitle, scaleTitle, level, icon, title	update	apiURL/devices/:deviceId/command/update
toggleButton	level, icon, title	on	apiURL/devices/:deviceId/command/on
camera	icon, title	depends on installed camera - could be: zoomIn, zoomOut, up, down, left, right, close, open	apiURL/devices/:deviceId/zoomIn
switchControl	level, icon, title,	change on, off, upstart, upstop, downstart, downstop, exact with get-param level	apiURL/devices/:deviceId/command/on
text	title, text, icon	-	-
sensorMultiline	multilineType, title, icon, level, (scaleTitle, ...)	depends on apps	apiURL/devices/:deviceId/command/:cmd
switchRGB	icon, title, color: r:255,g:255,b:255,	level on, off, exact with get-params: red, green and blue	apiURL/devices/:deviceId/command/exact? red=20&green=240&blue=0

Table 12.1: vDev device types with metrics and commands

ID refers to the command class the function is embedded in. The scale id is usually '0' unless the virtual device is generated from a Z-Wave device that supports multiple sensors with different scales in one single command class.
Virtual devices not generated by a Z-Wave device may have other Ids. They are either created by other physical device subsystems such as 433MHz or EnOcean or they are generated by a module (app).

12.3.2 Device Type

Virtual devices can have a certain types. Table shows the different types plus the defines commands. Table 12.1 shows the list of current device types with their metrics and defines commands.

12.3.3 Access to Virtual Devices

Virtual devices can be access both on the server side using JS modules and on the client side using the JSON API. On the client they are encoded into a URL style for easier handling in AJAX code. A typical client side command in the vDev API looks like

```
http://YOURIP:8083/ZAutomation/api/v1/devices/ZWayVDev_6:0:37/command/off
```

'api' points to the vDev API function, 'v1' is just a constant to allow future extensions. The devices are referred by a name that is automatically generated from the Z-Wave Device API. The vDev also unifies the commands 'command' and the parameters, here 'off'.
On the server side the very same command would be encoded in a JavaScript style.

Listing 1. Access vDevs

```
vdevId = vdev.id;

vDev = this.controller.devices.get(vdevId);

vDevList = this.controller.devices.filter(function(x) {
return x.get("deviceType") === "switchBinary"; });

vDevTypes = this.controller.devices.map(function(x) {
return x.get("deviceType"); });
```

12.3.4 Virtual Device Usage / Commands

In case the virtual device is an actor it will accept and execute a command using the syntax:

Vdev.performCommand(„name of the command“) The name of the accepted command should depend on the device type and can again be defined free of restrictions when implementing the virtual device. For auto-generated devices derived from Z-Wave the following commands are typically implemented.

1. 'update': updates a sensor value
2. 'on': turns a device on. Only valid for binary commands
3. 'off': turns a device off. Only valid for binary commands
4. 'exact': sets the device to an exact value. This will be a temperature for thermostats or a percentage value of motor controls or dimmers

12.3.5 Virtual Device Usage / Values

Virtual devices have inner values. They are called metrics. A metric can be set and get. Each virtual device can define its own metrics. Metrics can be level, title icon and other device specific values like scale (%, kWh, ...)

Listing 2.

```
vDev.set("metrics:...", ...);
vDev.get("metrics:...");
```

12.3.6 How to create your own virtual devices

A Virtual Device (Vdev) is an instance of a VirtualDevice class' descendant which exposes set of metrics and commands (according to it's type/subtype). Virtual devices are the only runtime instances which is controllable and observable through the JS API.
Technically, VDev is a VirtualDevice subclass which concretize, overrides or extends superclass' methods.

Step 1. Define a VirtualDevice subclass

Listing 3.

```
// Important: constructor SHOULD always be successful
BatteryPollingDevice = function (id, controller) {
    // Always call superconstructor first
    BatteryPollingDevice.super_.call(this, id, controller);

    // Define VDevs properties
    this.deviceType = "virtual";
    this.deviceSubType = "batteryPolling";
    this.widgetClass = "BatteryStatusWidget";

    // Setup some additional metrics (many of them is setted up in a base class)
    this.setMetricValue("someMetric", "someValue");
}
inherits(BatteryPollingDevice, VirtualDevice);
```

VDev class should always fill in the deviceType property and often fill in the deviceSubType property.
If the particular VDev class can be controller by the client-side widget, it should define widget's class name in the widgetClass property.

Step 2. Override performCommand() method

Listing 4.

```
BatteryPollingDevice.prototype.performCommand = function (command) {
    var handled = true;
```

```
    if ("update" === command) {
        for (var id in zway.devices) {
            zway.devices[id].Battery && zway.devices[id].Battery.Get();
        }
    } else {
        handled = false;
    }

    return handled ? true :
    BatteryPollingDevice.super_.prototype.performCommand.call(this, command);
}
```

VDev itself mostly needed to handle commands, triggered by the events, system or the API.
In the example above you could see, that this VDev is capable of performing "update" command. But base class can be capable of performing some other commands, so the last l ine calls superclass' performCommand() method if the particular command wasn't handled by the VDev itself.
This extensibility provides the possibility to create a VDev class tree. Take a look at ZWaveGate module as an example of such tree.

Step 3. Instantiate your VDev by the module

Listing 5.

```
// ...part of the BatteryPolling.init() method
executeFile(this.moduleBasePath()+"/BatteryPollingDevice.js");
this.vdev = new BatteryPollingDevice("BatteryPolling", this.controller);
```

First line of code is loads and executes apropriate .js-file which provides BatteryPollingDevice class.
Secnd line instantiates this class.
The last line calls controller's registerDevice method to register and VDev instance.

Step 4. Register device

Listing 6. Register Device

```
        vDev = this.controller.devices.create(vDevId, {
            deviceType: "deviceType",
            metrics: {
                level: "level",
                icon: "icon from lib or url"
                title: "Default title"
            }
        }, function (command, ...) {
                // handles actions with the widget
        });
```

Step 5: Unregister device

Devices can be deleted or unregistered using the following command:

this.controller.devices.remove(vDevId)

12.3.7 Binding to metric changes

The metric - the inner variables of the vDev a changed by the system automatically. In order to perform certain functions on these changes the function needs to be bound to the change to the vdev. The syntax for this is

vDev.on('change:metrics:...", function (vDev) ...); Unbinding then works as one can expect:

vDev.off('change:metrics:...", function (vDev) ...)

12.4 The event bus

All communication from and to the automation modules is handled by events. An event is a structure containing certain information that is exchanged using a central distribution place, **the event bus**. This means that all modules can send events to the event bus and can listen to event in order to execute commands on them. All modules can 'see' all events but need to filter out their events of relevance. The core objects of the automation are written in JS and they are available as source code in the sub folder 'classes':

- AutomationController.js: This is the main engine of the automation function
- AutomationModule.js: the basic object for the module

The file main.js is the startup file for the automation system and it is loading the three classes just mentioned. The subfolder /lib contains the key JS script for the Event handling: eventemitter.js.

12.4.1 Emitting events

The 'Event emitter' emits events into the central event bus. The event emitter can be called from all modules and scripts of the automation system. The syntax is:

$$controller.emit(eventName, args1, arg2, ...argn)$$

The event name 'eventName' has to be noted in the form of 'XXX.YYY' where 'XXX' is the name of the event source (e.g. the name of the module issuing the event or the name of the module using the event) and 'YYY' is the name of the event itself. To allow a scalable system it makes sense to name the events by the name of the module that is supposed to receive and to manage events. This simplifies the filtering of these events by the receiver module(s).

Certain event names are forbidden for general use because they are already used in the existing modules. One example are events with the name cron.XXXX that are used by the cron module handling all timer related events.

Every event can have a list of arguments developers can decide on. For the events used by preloaded modules (first and foremost the cron module) this argument structure is predefined. For all other modules the developer is free to decide on structure and content. It is also possible to have list fields and or any other structure as argument for the event

One example of an issued event can be

$$emit(\text{ı}mymodule.testevent\text{ȷ}, \text{ȷ}Test\text{ȷ}, [\text{ı}event1\text{ȷ}, \text{ȷ}event2\text{ȷ}])$$

12.4.2 Catching (binding to) events

The controller object, part of every module, offers a function called 'on()' to catch events. The 'on(name, function())' function subscribes to events of a certain name type. If not all events of a certain name tag shall be processed a further filtering needs to be implemented processing the further arguments of the event. The function argument contains a reference to the implementation using the event to perform certain actions. The argument list of the event is handed over to this function in its order but need to be declared in the function call statement.

```
this.controller.on("mymodule.testevent", function (name,eventarray) )
```

The same way objects can unbind from events:

```
this.controller.off("mymodule.testevent", function (name,eventarray));
```

12.4.3 Notification and Severity

Notifications are a special kind of event to inform the user on the graphical user interface or out-of-band.. This means that normal events are typically describes with numbers or ids while notifications contain a human readable message. The UI can be notified on the certain events.

this.controller.addNotification("....severity....", "....message....", "....origin...."); The parameters define

- severity is error, info, debug;
- origin describes which part of the system it is about: core, module, device, battery.

The controller can act on notifications or disable them.

this.controller.on('notifications.push', this.handler);

this.controller.off('notifications.push', this.handler);

12.5 Modules (for users called 'Apps')

Beside the core functions encoded into the JS core there are extensions to this code called modules. Modules extend the JS core by providing internal or external (visible to the user) functions.
Each modules code is located in a sub directory of the sub folder module as described in chapter 11.4.1. The name of the subfolder equals the name of the module. The sub folder contains files to define the behavior of the module.

```
ModuleName
 ├─ Module.json:  The Manifest file of the Module
 ├─ index.js:  The main JS file
 ├─ htdocs:  ressources accessible by the web server
 └─ lang:  translation into local languages
```

12.5.1 Module.json

This file contains the module meta-definition used by the AutomationController. It must be a valid JSON object with the following fields (all of them are required):

- **autoload** — Boolean, defines will this module automatically instantiated during Home Automation startup.
- **singleton** — Boolean, defines this module can be instantiated more than one time or not.
- **defaults** — Object, default module instance settings. This object will be patched with the particular config object from the controller's configuration and resulting object will be passed to the initializer.
- **actions** — Object, defines exported module instance actions. Object keys are the names of actions and values are meta-definitions of exported actions used by AutomationController and API webserver.
- **metrics** — Object, defines exported module metrics.

All configuration fields are required. Types of the object must be equal in every definition in every case. For instance, if module doesn't export any metric corresponding key value should be and empty object "".

12.5.2 index.js

This script defines an automation module class which is descendant of AutomationModule base class. During initialization the module script must define the variable '_module' containing the particular module class.
Example of a minimal automation module:

Listing 1. Minimal Module

```
function SampleModule (id, controller) {
SampleModule.super_.call.init(this, id, controller);

this.greeting = "Hello,World!";
}

inherits(SampleModule, AutomationModule);
_module = SampleModule;

SampleModule.prototype.init = function () {
    this.sayHello();
}

SampleModule.prototype.sayHello = function () {
    debugPrint(this.greeting);
}

SampleModule.prototype.stop = function () {
    this.sayByeBye();
}
```

The first part of the code illustrates how to define a class function named SampleModule that calls the superclass' constructor. Its highly recommended not to do further instantiations in the constructur. Initializations should be implemented within the 'init' function.
The second part of the code is almost immutable for any module. It calls prototypal inheritance support routine and it fills in _module variable.
The third part of the sample code defines module's init() method which is an instance initializer. This initializer must call the superclass's initializer prior to all other tasks. In the initializer module can setup it's private environment, subscribe to the events and do any other stuff. Sometimes, whole module code can be placed withing the initializer without creation of any other class's methods. As the reference of such approach you can examine AutoOff module source code.
After the init function a module may contain other functions. The 'sayHello' function of the Sample Module shows this as example.

12.5.3 Available Core Modules

All modules in Z-Way are designed the same way using the same file structure but they serve different purposes and they are of different importance:

- Core Module are modules that provide essential parts of the Z-Way system. They run from the beginning and should not be terminated without good reason. Normal users will not even see them in the list of active apps. Users with management privilege can set a checkbox in their My Settings to unhide them.
- Standard Modules can be started and stopped by the user. they are already in the subfolder 'modules' and can not be deleted.
- Modules from the Online Service must be downloaded first before they can be used. They can be started, stopped and even removed. They are stored in the folder 'userModules'.

The two core modules are worth to be explained in detail:

Cron, the timer module

All time driven actions need a timer. The Z-Way automation engine implement a cron-type timer system as a module as well. The basic function of the cron module is

- It accepts registration of events that are triggered periodically
- It allows to de-register such events.

The registration and deregistration of events is also handled using the event mechanism. The cron module is listening for events with the tags 'cron.addTask' and 'cron.removeTask'. The first argument of these events are the name of the event fired by the cron module. The second argument of the 'addTask' event is an array desricing the times when this event shall be issued. It has the format:

- Minute [start,stop, step] or 0-59 or null
- Hour [start,stop, step] or 0-23 or null
- weekDay [start,stop, step] or 0-6 or null
- dayOfMonth [start,stop, step] or 1-31 or null
- Month [start,stop, step] or 1-12 or null

The argument for the different time parameters has one of three formats

- null: the event will be fired on every minute or hour etc.
- single value: the event will be fired when the value reaches the given value
- array [start, stop, step]: The event will be fired between start and stop in steps.

The object

$$\{minute : null, hour : null, weekDay : null, day : null, month : null\}$$

will fire every minute within every hour within every weekday on every day of the month every month. Another example of an event emitted towards the cron module for registering an timer event can be found in the Battery Polling Module:

Listing 2. Registering a Battery Polling Command

```
this.controller.emit("cron.addTask", "batteryPolling.poll", {
    minute: 0,
    hour: 0,
    weekDay: this.config.launchWeekDay,
    day: null,
    month: null
});
```

This call will cause the cron module to emit an event at night (00:00) on a day that is defined in the configuration variable this.config.launchWeekDay, e.g. 0 = Sunday.
The 'cron.removeTask' only needs the name of the registered event to deregister.

Z-Wave

The whole mapping of Z-Wave devices into virtual devices is handled by a module called **'ZWave'**. This module is quite powerful. It does not only manage the mapping but handles various Z-Wave specific functions such as timing recording, etc.

13 Special topics for Developers

13.1 Authentication

In order to access API one need to authenticate itself. Z-Way uses sessions to authenticate users. Z-Way session is called **ZWAYSession**.
If remote access is used, the find.z-wave.me service will issue an additional token called **ZBW_SESSID**.
Please read in 13.1.4 about the token lifetime, permanent tokens and token usage guide before selecting your preferred authentication method.

13.1.1 Local authentication

If used in local network, Z-Way can be directly addressed via

```
http://IP:8083
```

Note that if Z-Way is used in trusted network and only Z-Wave API is supposed to be used, you might consider to skip authentication and check option *Public API access* in Z-Wave Binding app.
Z-Way session can be obtained by sending login and password in JSON format using POST to URL

```
/ZAutomation/api/v1/login
```

User credentials should look like `{"login":"admin", "password":"admin"}`.
It is also possible to use Basic Authentication to transmit login and password.
In return the session will be sent in three forms:

- as **data.sid** field in JSON structure (only for login using JSON),
- as a header called **ZWAYSession**,
- as a cookie called **ZWAYSession**.

Example of successful login will look like:

Listing 1. Successful login reply

```
{
    "data": {
        "sid": "ba69cb5b-b2fd-5ce0-5b75-9bae3e8bc369",
        "id": 1,
        "role": 1,
        "name": "Administrator",
        "lang": "en",
        "color": "#dddddd",
        "dashboard": [],
        "interval": 2000,
        "rooms": [
            0
        ],
        "hide_all_device_events": false,
        "hide_system_events": false,
        "hide_single_device_events": []
    },
    "code": 200,
    "message": "200 OK",
    "error": null
}
```

Listing 2. Wrong login/password reply

```
{
    "data": null,
    "code": 401,
```

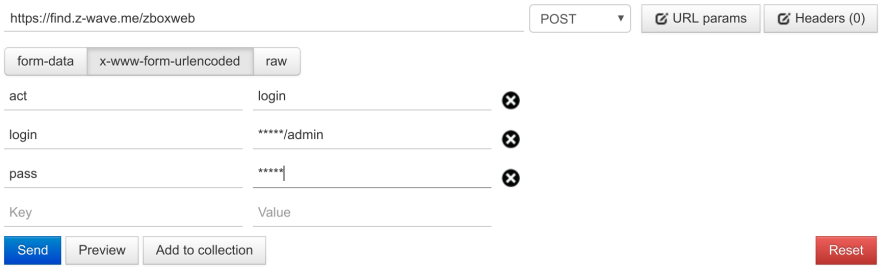


Figure 13.1: Example of login on find.z-wave.me using Postman

```
    "message": "401 Unauthorized",
    "error": "User login/password is wrong."
}
```

Once obtained, the session can be sent to the Z-Way server via the following ways:

- as an HTTP header called **ZWAYSession**,
- as a cookie called **ZWAYSession**,
- as an HTTP Authentication Bearer header (see 13.1.4)

Below are few examples of local authentication.
With cURL using cookies:

```
curl -H "Accept: application/json" -H "Content-Type: application/json" -X POST -d '{"form": true, "login": "admin", "password": "admin"}' http://192.168.0.62:8083/ZAutomation/api/v1/login -c cookie.txt
curl http://192.168.0.62:8083/ZAutomation/api/v1/... -b cookie.txt
```

With cURL using Basic Authentication:

```
curl -u admin:admin http://192.168.0.62:8083/ZAutomation/api/v1/...
```

With wget using Basic Authentication:

```
wget –auth-no-challenge –user=admin –password=pwd 192.168.0.62:8083/ZAutomation/api/v1/...
```

From the browser using jQuery and Basic Authentication:

Listing 3. Login with jQuery

```
jQuery('img').click(function() {
    jQuery.ajax({
        url: "http://192.168.0.62:8083/ZAutomation/api/v1/...",
        beforeSend: function (xhr) { xhr.setRequestHeader ("Authorization", "Basic " + btc
    });
});
```

13.1.2 Remote authentication

If not disabled by the user, Z-Way provides remote access service to the controller via

`https://find.z-wave.me`

See section 3.3 for more information on the Z-Way remote access service.
This service accepts **Z-Way Platform ID** , username and password, checks entered credentials agains your Z-Way and if accepted returns back two sessions **ZWAYSession** and **ZBW_SESSID** in two forms:

- as a header,
- as a cookie.

The session can be obtained by

- logging in the form on

  `https://find.z-wave.me/zboxweb`

- sending a POST request to

  `https://find.z-wave.me/zboxweb`

  with form data `act=login&login=id/admin&pass=password`

Once logged in with `act=login`, the user is redirected to the / of the Z-Way (in most cases it then redirects to /smarthome/). If you want the user to be redirected to some specific URL after log in add one more parameter `ruri=...` (see examples below).

The obtained session will be returned

- as an HTTP header,
- as a cookie.

To suppress the redirect use `act=auth`.

The credential can be used in subsequent requests in one of the three forms listed below (in order they are checked in Z-Way):

- as HTTP **Authentication: Bearer** header (see below),
- as HTTP **X-ZBW_SESSID** and **ZWAYSession** headers,
- as **ZBW_SESSID** and **ZWAYSession** cookies.

Below are examples of remote authentication.

With cURL using cookies:

```
curl -H "Accept: application/json" -X POST -d 'act=login&login=132339/admin&pass=admin1' https://find.z-wave.me/zboxweb -c cookie.txt
curl https://find.z-wave.me/ZAutomation/api/v1/... -b cookie.txt
```

If you want the user to be redirected to a specific page after login, you can also use the following url:

`https://find.z-wave.me/zboxweb?ruri=/expert`

or

`https://find.z-wave.me/zboxweb?ruri=/expert&id=123456&login=admin`

to pre-fill **Z-Way Platform ID** and user name.

13.1.3 Remote authentication and access error handling

The tables below describe possible errors returned by the remote access service. Errors are order exactly as the request is handled.

Authentication on `https://find.z-wave.me/zboxweb` with `act=login`:

Condition	HTTP code	Redirect URL
Box not connected to the server	302	`https://find.z-wave.me/zboxweb?err=bad_login&ruri=%2F`
Box connected, but default login/password admin/admin is used	302	`https://find.z-wave.me/zboxweb?err=insecure_login_pass&ruri=%2F`
Box not connected to find.z-wave.me	502	
Box connected, but login/password is invalid	302	`https://find.z-wave.me/zboxweb?err=bad_login&ruri=%2F`
Box connected and authentication is successful	302	`https://find.z-wave.me/`

Authentication on `https://find.z-wave.me/zboxweb` with `act=auth`:

Condition	HTTP code	HTTP error message
Box not connected to the server	403	Forbidden Wrong username or password
Box connected, but default login/password admin/admin is used	403	insecure_login_pass Your (username, pass) pair is insecure
Box not connected to find.z-wave.me	502	
Box connected, but login/password is invalid	403	Forbidden Wrong username or password
Box connected and authentication is successful	200	

Subsequent requests to `https://find.z-wave.me/PATH`, where `PATH` is not equal to `zboxweb` or `zboxweb/*`:

Condition	HTTP code	Redirect URL
No token (no Authorization header, no X-ZBW-SESSID header, no ZBW_SESSID cookie)	307	`https://find.z-wave.me/zboxweb/r//PATH`
Invalid token (incorrect or expired or revoked)	307	`https://find.z-wave.me/zboxweb/r//PATH`
Box not connected to find.z-wave.me	502	
Server side error	500	

13.1.4 Token lifetime

Each time you log in using methods described above a new session is created. This session will live for one week (might be changed on Z-Way side and on find.z-wave.me side, do not rely on this period).

Those login methods are good to obtain the token once to the access the API via token or for rare single actions. Creating a new session each time will pollute user profile in Z-Way.

Tokens can be deleted at any time in User management (4.3.1) panel. This will lead to log-out of the application that is using this token. This application will loose access to Z-Way and will require again a log in via username and password.

To use tokens long term we recommend to make them permanent. You can make any token permanent by pressing a button in User management (4.3.1). Permanent token do never expire until deleted. Only Z-Way token can be made permanent that way.

To make both find.z-wave.me and Z-Way token permanent use **Authentication Bearer** way to transmit tokens.

Both find.z-wave.me remote access service and Z-Way do support **Authentication Bearer** HTTP header. This header is commonly used in OAuth2 authentication protocol, but you can use it without OAuth2 too.

Z-Way and find.z-wave.me will automatically mark tokens as permanent if they are received in **Authentication Bearer** header.

The format of the header is: `Authorization: Bearer` **`ZBW_SESSID`**`/`**`ZWAYSession`**. Note that if direct access is needed (not via find.z-wave.me), **ZBW_SESSID** can be omitted. In this case a slash / should still preceed **ZWAYSession**

A typical application should take usename and password to log in via find.z-wave.me and get both **ZWAYSession** and **ZBW_SESSID** tokens to form Authentication Bearer token. The username and password should then be erased from the memory, while Authentication Bearer token should be saved for future use. It is not recommended to save user credentials.

Like other tokens user can delete the permanent token in the User management (4.3.1) panel.

13.1.5 OAuth2

Z-Way do also support OAuth2 authentication method to provide access to smart home devices to third party services like voice assistances (Amazon Alexa, Google Home, Yandex Alice), IFTTT and some others.

To get integrated one should provide to the Z-Wave.Me team redirect_uri and get back OAuth2 server URL, client_id and client_secret. Please contact info@z-wave.me.

To let the user authenticate and grant access to his devices you should redirect him to one of the two pages:

```
https://z-wave.me/oauth2/?state=...&redirect_uri=https://...&response_type=
code&client_id=...
https://find.z-wave.me/zboxweb?lang=...&hide_diruris=1&ruri=/smarthome/%23/
oauth2%3Fstate%3D...%26redirect_uri%3Dhttps%3A%2F%2F...%26response_type%
3Dcode%26client_id%3D...
```

In this request:

- lang (optional — English if omitted) — pre-selects the find.z-wave.me interface language (see ZBW_IFLANG cookie for the correct value)
- ruri — URL encoded link to the target page in Z-Way user interface to OAuth2 page:

  ```
  /smarthome/#/oauth2?state=...&redirect_uri=https://...&response_type=
  code&client_id=...
  ```

  - state — specific value used by your service to match the OAuth2 reply with the original request — used

with redirected to redirect_uri
- redirect_uri — the URL to redirect the user to once the authorization was granted
- response_type — the type of OAuth2 authorization; only *code* is currently supported
- client_id — id of the client that should match redirect_uri on Z-Wave.Me OAuth2 server.

After the user logs in to Z-Way he is asked to mark rooms and devices to be granted access to. A new user will then be created with the corresponding permissions and a session token will be generated. The session is saved on the Z-Wave.Me OAuth2 server together with an authorization code.

On the next step the user is redirected back to the service using URL

```
redirect_uri?state=...&code=...
```

In this request:

- state — specific value used by your service to match the OAuth2 reply with the original request (see above)
- code — the authorization code to be used to get access token from the OAuth2 server

The service can now make a POST request to the Z-Wave.Me OAuth2 server on

```
/token
```

providing `client_id`, `client_secret` and `code`. The request can use JSON format or form data or url encoded form. The response is a JSON with access_token or error message.

The resulting access_token is to be used in the Authentication Bearer HTTP header. This token never expires unless deleted by the user.

Listing 4. Request of Access token via Authorization code (JSON format)

```
Content-type: application/json
{
    "client_id": "...",
    "client_secret": "...",
    "code": "..."
}
```

Listing 5. Request of Access token via Authorization code (URL encoded form)

```
Content-Type: application/x-www-form-urlencoded
client_id=...&client_secret=...&code=... (URL encoded!)
```

Listing 6. Request of Access token via Authorization code (form data)

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarytufRVLxOz9VdsQbA

------WebKitFormBoundarytufRVLxOz9VdsQbA
Content-Disposition: form-data; name="client_id"

...
------WebKitFormBoundarytufRVLxOz9VdsQbA
Content-Disposition: form-data; name="client_secret"

...
------WebKitFormBoundarytufRVLxOz9VdsQbA
Content-Disposition: form-data; name="code"

...
------WebKitFormBoundarytufRVLxOz9VdsQbA--
```

Listing 7. Access token successfuly retrieved

```
{
    "access_token": ".../...",
    "token_type": "bearer"
}
```

Listing 8. Authorization code is invalid (400 Bad request)

{ "singleton" : false, "dependencies": [],	Boolean to set if there can be multiple Instances of the module allowed or not An array list of all module names from which this module is dependent. Modules in this list should be 'singleton'.Thew module cannot be instantiated if at least one of the modules in the list does not have an instance.
"category": "automation_basics",	The app category this module is shown in the app store. Known app store categories are: 'basic_gateway_modules', 'legacy_products_workaround', 'support_external_ui', 'support_external_dev', 'automation_basic', 'device_enhancements', 'developers_stuff', 'complex_applications', 'automation', 'security', 'peripherals', 'surveillance', 'logging', 'scripting', 'scheduling', 'climate', 'environment', 'scenes', 'notifications', 'tagging'
"author": "Z-Wave.Me",	Author name of the Module
"homepage": "http://razberry.z-wave.me",	If you have a news homepage, it can be linked here.
"icon": "icon.png",	Name of the icon which is shown for this module on the UI
"moduleName": "AppClassName",	Module name have to the same like the class reference
"version": "1.0.0",	Version number of this module
"maturity": "beta",	Status if the app is still in development or released
"repository": { Repository optional "type": "git", Kind of the repository "source": https://github.com/ZWaveMe/homeautomation },	Address of the repository
"defaults" : { "title" : "__m_title__",	The title placeholder for the Language files
"description" : { "_m_descr__" },	The description placeholder for the language files
"schema" : {},	Description of the data structure of the form for instantiating the module. See explanation of schema for details
"options" : {}}	Showing options of the setup form
"description" : { "_m_descr__" },	The description placeholder for the language files

Table 13.1: Module.json details

```
{
    "error": "invalid_grant"
}
```

Listing 9. Wrong credentials (400 Bad request)

```
{
    "error": "invalid_client"
}
```

13.2 How to write own Apps for Z-Way

According to Chapter 12.5 apps have two core files:

- module.js
- index.js

The following chapter explains these two files more in detail.

13.2.1 module.js

Module.js defines the general behavior of the app and the interface to the user side. Table 13.1 shows the structure of the file module.js with an explanation of each line item.

13.2.2 Schema

The **schema** is a JSON Structure to define the user interface of the module. It lists all input parameters and options to be shown in the setup dialog of the app:

Listing 1. Schema Structure

```
{
  "schema": {
    "type": "object",
    "properties": {
    }
  }
}
```

The structure of the schema is the following. Inside the 'properties' space the single 'properties' can be defined. They become the parameter of the module during the initiation and they are shown as configuration parameters in the setup dialog. There are different types of input parameters:

Primitive data types like integer, float or string

Listing 2. Schema Structure Simple Type

```
{
  //Parametername
  "name": {
    "type": "array",
    "items": {
      "title": "Device",
      "type": "radio",
      //array of choosable items
      "enum": ["Adult", "Child"],
      "default": "Child",
      "required": true
    }
  },
  //Parametername
  "name": {
    "type": "integer",
    "required": true
  },
}
```

Name Spaces - Enumerations with a choice

Listing 3. Schema Structure Enumerations with a choice

```
{
  //Parametername
  "name": {
    "field": "enum",
    "datasource": "namespaces",
    //special namespacedestination
    "enum": "namespaces:devices\_all:deviceId",
    "required": true
  },
}
```

Name spaces refer to the internal Z-Way structure. It allows to list elements from the Z-Way data model and filter it. The statement `"namespaces:devices_all:deviceId"` will offer a selection of all devices.
Namespaces can also be combined like

`function AppClassName (id, controller) AppClassName.super_.call(this, id, controller);`	Constructor method: This line is a call of the Supercon-structor. It has always to be first line of the constructor
`inherits(AppClassName, AutomationModule);`	inheration call:
`_module = AppClassName;`	The definition of the class reference
`AppClassName.prototype.init = function (config) AppClassName.super_.prototype.init.call(this, config); var self = this; ;`	Initialization method: Variable to refer to in the class in own methods (this is context dependent in JavaScript). Here you can register 'listeners' for the event bus. For details on event bus please refer to chapter 12.4
`AppClassName.prototype.stop = function () AppClassName.super_.prototype.stop.call(this);;`	Destroy method: Here you have to unregister 'listeners'.
`AppClassName.prototype.Methodname= function(parameter)`	Own Methods: Write your own Methods here.

Table 13.2: Details of index.js

`namespaces:devices_doorlock:deviceId,namespaces:devices_switchBinary:deviceId`
which means devices doorlock and all binary switches. Namespaces can also be REST paths like
`server:port/v1/namespaces/{devices_DEVICETYPE}.{PATH}`

13.2.3 The file index.js

Thew file index.js contains the application as such. It can include other js files is needed but Z-Way will always look for a index.js file to load first. Table 13.2 list the basic structure of index.js with the minimum functions.
More information e.g. about the list of probe types etc. you find on
`http://docs.zwayhomeautomation.apiary.io/`

13.3 Write you own Device Description Files

This part of the manual is not yet published because the service for creating own Device Description Files is not yet available.

13.4 Extending EnOcean

How to include a new EnOcean Device (example Hoppe Door handle)
(1) Check if the profile is in **/opt/z-way-server/config/Profile.xml**
Not that you just need to know the profile the given product supports. There is no way to find out automatically!

Listing 1. EnOcean Profile Entry

```
<Profile rorg="0xf6" func="0x10" type="0x00" rorgDescription="RPS Telegram"
    funcDescription="Mechanical Handle" typeDescription="Window Handle">
  <Field offset="0" size="4" name="windowHandle" type="int" description="Movement of the w
</Profile>
```

(2) Add the device record of the device to
/opt/z-wave-server/htdocs/smarthome/storage/data/devices_encoean.json.
Here the rorg, funcid and type are set. Now the device record will be created and the right values are changed on message reception. Now you need to make sure the right element is rendered and updated. This is in **/opt/z-wave-server/automation/**
First add a filter to catch the events and call the correct function:

Listing 2. Catch Device IDs

```
if (matchDevice(0xf6, 0x10, 0x01)) {
  // Hoppe Window handle
  windowHandle("contact", "window", "Windor Handle");
}
```

now you add the function that handles the value changes and renders the element accordingly. For the window handle we use the binary sensor element but overwrite the status information according to the information of the window handle moves.

Listing 3. Handle Device

```
function windowHandle(dh, type, title) {
var vDev = self.controller.devices.create({
deviceId: vDevIdPrefix + type,
defaults: {
deviceType: 'sensorBinary',
metrics: {
probeTitle: type,
scaleTitle: '',
icon: type,
level: '',
title: title
}
},
overlay: {},
handler: function(command) {},
moduleId: self.id
});

if (vDev) {
self.dataBind(self.gateDataBinding, self.zeno, nodeId, dh,
function(type) {
try {
if (this. handleValue == 13)
vDev.set("metrics:level", "tilt");
if (this. handleValue == 15)
vDev.set("metrics:level", "closed");
if (this. handleValue == 12 || this. handleValue == 14)
vDev.set("metrics:level", "open");
} catch (e) {}
}, "value");
}
}
```

A Z-Way Data Model Reference

A.1 Data

This is the description of the data object (called Data Holder or DH).
General note: Z-Way objects and it's decendents are NOT simple JS objects, but native JS objects, that does not allow object modification.

- name: Name of data tree element
- updated: Update time
- invalidated: Invalidate time
- valueOf(): Returns value of the object (can be omitted to get object value)
- invalidate(): Invalidate data value (mark is as not valid anymore)
- bind(function (type[, arg]) ..., [arg, [watchChildren=false]]): Bind function to a change of data tree element of its descendants
- unbind(function): Unbind function bind previously with bind()

A.2 JS object zway

- Description: zway is the Z-Wave part of the object tree
- Syntax: zway.X with X as child object
- Child objects
  - controller: controller object, see below for details
  - devices: devices list, see below for details
  - version: Z-Way.JS version
  - isRunning(): Check if Z-Way is running
  - isIdle(): Check if Z-Way is idle (no pending packets to send)
  - discover(): Start Z-Way discovery process
  - stop() : Stop Z-Way
  - InspectQueue() : Returns list of pending jobs in the queue.
    - item: [timeout, flags, nodeId, description, progress, payload]
    - flags: [send count, wait wakeup, wait security, done, wait ACK, got ACK, wait response, got response, wait callback, got callback]
  - ProcessPendingCallbacks(): Process pending callbacks (result of setTimeout/setInterval or functions called via HTTP JSON API)
  - bind(function, bitmask): Bind function to be called on change of devices list/instances list/command classes list
  - unbind(function) : Unbind function previously bind with bind()
  - all function classes in C are also methods of this data object

A.3 controller

You can access the data elements of "controller" in the **Z-Wave Expert User Interface** in menu `Network > Controller Info` using the buttons Show controller data and Show controller's device data.

- Description: Controller object
- Syntax: controller.X with X as child object
- Child objects
  - data: Data tree of the controller
    - homeId: Home ID of the controller
    - nodeId: Node ID of the controller
    - SISPresent: is SIS available (if TRUE, SUCNodeId is a SIS, otherwise it is SUC)
    - SUCNodeId: Node ID of SUC or SIS or 0 if no SUC/SIS present
    - isInOtherNetworks: is controller the original Primary or it is in other's network
    - isPrimary: can controller include devices (Primary or Inclusion controller)
    - isRealPrimary: is controller Primary Controller or SIS in the network

    - isSUC: is SUC present
    - libType: Z-Wave library type
    - frequency: current frequency of the transceiver
    - controllerState: current network management state of the controller
    - lastExcludedDevice: Node ID of last excluded device
    - lastIncludedDevice: Node ID of last included device
    - secureInclusion: shall inclusion be done using Security
    - caps: Z-Way license information
    - softwareRevisionVersion: version of Z-Way build
    - softwareRevisonDate: date of Z-Way build
    - softwareRevisionId: git commit of Z-Way build
    - manufacturerId / manufacturerProductId / manufacturerProductTypeId: IDs to identify the transceiver hardware
    - vendor: name of hardware vendor
    - APIVersion: Version of the Serial API of the transceiver firmware
    - SDK: Z-Wave SDK version of the transceiver firmware
    - ZWVersion: ZWave Version (firmware)
    - ZWaveChip: Serie of the transceiver Z-Wave chip
    - ZWlibMajor / ZWlibMinor: library version
    - capabilities: array of Function Class IDs supported by the transceiver firmware
    - functionClasses: ordered array of IDs of Function Classes
    - functionClassesNames: ordered array of Names of Function Classes
    - uuid: Z-Way transceiver firmware unique ID
    - memoryGetAddress: address of last data stored in memoryGetData read by one of memory read function
    - memoryGetData: last data read by one of memory read function
    - countJobs: shall job be counted (nonManagementJobs and devices[x].data.queueLength)
    - nonManagementJobs: number of non-management jobs in the queue
    - deviceRelaxDelay: time in 10ms to wait before sending next command to same device (configurable in Defaults.xml)
    - promiscMode: true if Zniffer promiscuous mode is enabled
    - incomingPacket: last incoming packet from Z-Wave network
    - outgoingPacket: last outgoing packet to Z-Wave network
    - curSerialAPIAckTimeout10ms: timing parameter of Serial API
    - curSerialAPIBytetimeout10ms: timing parameter of Serial API
    - oldSerialAPIAckTimeout10ms: previous timing parameter of Serial API
    - oldSerialAPIBytetimeout10ms: previous timing parameter of Serial API

Function classes as shown in section C are called as methods of the object zway.controller.

A.4 Devices

The devices object contains the array of the device objects. Each device in the network - including the controller itself - has a device object in Z-Way.

- Description: list of devices
- Syntax: X with X as child object
- Child objects
  m : Device object
  - length: Length of the list
  - SaveData(): Save Z-Way Z-Wave data for hot start on next run (in config/zddx/HOMEID-DevicesData.xml)

A.5 Device

The data object can be accesses in the **Z-Wave Expert User Interface** in advanced mode of 'Configuration'

- Description: the device object
- Syntax: device[n].X with X as child object
- Child objects
  - id: (node) Id of the device
  - Data: Data tree of the device
    - SDK: SDK used in the device firmware

    - ZDDXMLFile: file of the Devcie Description Record
    - ZWLib: Z-Wave library used in the device firmware
    - ZWProtocolMajor / ZWProtocolMinor: Z-Wave protocol version
    - applicationMajor / ApplicationMinor: Application Version of devices firmware
    - manufacturerId / manufacturerProductId / manufacturerProductTypeId: ids used to identify the device
    - basicType: Z-Wave Basic Type
    - genericType: Z-Wave Generic Type
    - specificType: Z-Wave Specific Type
    - deviceTypeString: verbal Z-Wave Device Class
    - vendorString: verbal vendor name
    - nodeInfoFrame: Node Information Frame (NIF) array
    - isListening: is always listening
    - isAwake: is currently awake
    - keepAwake: shall the device be kept awake even if there is nothing to send to it
    - isRouting: is abale to send routed unsolicited packets
    - sensor1000: device is a FLiRS with 1000 ms wakeup
    - sensor250: device is a FLiRS with 250 ms wakeup
    - isVirtual: is virtual device from a bridge controller
    - option: are optional Command Classes present in addition to mandatory for this Device Class
    - infoProtocolSpecific: internal information about the device
    - interviewDone: set to true when device finishes the interview (might not match with interviewDone on individual CCs if they appear after interview done)
    - neightbours: list of neighbour nodes
    - priorityRoutes: list of priority routes assigned to the node (via AssignPrioritySUCReturnRoute and AssignPriorityReturnRoute) used to restore routes on network reorganization
        - hop 1, hop 2, hop 3, hop 4, speed: route hops node Id (1..232 or 0 for end of route), speed flag
    - givenName: name for Expert UI
    - isFailed: is failed
    - failureCount: number of tries since last device failed
    - lastRecevied: timestamp of last packet received
    - lastSend: timestamp of last sent operation
    - lastPacketInfo: structure with deliveryTime, delivered and packetLength information about last packet sent
    - queueLength: length of device specific send queue (if countJobs is enabled)
    - lastNonceGet: internal
    - acceptSetSecurityLevel: minimal security level that is accepted for Set commands from the device: 0 - accept both secure S0/S2 or unsecure Set commands, 128 - accept S0 and all S2 levels, 1 - accept all S2 levels (Unauthenticated, Authenticated and Access), 2 - accept S2 Authenticated and S2 Access, 4 - accept only S2 Access
- instances: iInstances list of the device
- RequestNodeInformation(): Request NIF
- RequestNodeNeighbourUpdate(): Request routes update
- InterviewForce(): Purge all command classes and start interview based on device's NIF
- RemoveFailedNode(): Remove this node as failed. Device should be marked as failed to remove it with this function.
- SendNoOperation(successCallback = NULL, failureCallback = NULL): Ping the device with empty packet (even if device is not reachable successCallback is called - use isFailed to check device availability)
- LoadXMLFile(file): Load new Z-Wave Device Description XML file. See http://pepper1.net/zwavedb/
- GuestXML(): Return the list of all known Z-Wave Device Description XML files with match score. [score, file name, brand name, product name, photo]
- WakeupQueue(): Pretend the device is awake and try to send packets
- AssignReturnRoute(target): Send device new routes to target node
- DeleteReturnRoute(): Clear routes in device
- AssignSUCReturnRoute(): Inform device about SUC and route to reach it

A.6 Instances

Each device may have multiple instances (similar functions like switches, same type sensors, ...) If only one instance is present the id of this instance is 0. Command classes are located in instances only.

- Description: list of instances
- Syntax: device[n].instance[m].X with X as child object
- Child objects
  m : instance object
  - length: Length of the list
  - commandClasses: list of command classes of this instance. In case there is only one instance, this is equivalent to the list of command classes of the device. For details see below.
  - Data: data object of instance
    - dynamic: flag if instance is dynamic
    - genericType: generic Z-Wave device class of instance
    - specificType: specific Z-Wave device class of instance

A.7 CommandClass

This is the Command Class object. It contains public methods and public data elements that are described in chapter B

- Description: Command Class Implementation
- Syntax: device[n].instance[m].commandclass[id].X with X as child object
- Child objects
  - id: Id of the Command Class of the instance of the device
  - data: Data tree of the Command Class
    - interviewCounter: number of attempts left until interview is terminate even if not successful
    - interviewDone: flag if interview of the command class is finished
    - security: flag if Command Class is operated under Security Command Class
    - version: version of the Command Class implemented in the device
    - supported: flag if Command Class is supported or only controlled
    - commandclass data: Command Class specific data - see chapter B for details.
  - name: Command Class name
  - Method: Command Class method - see chapter B for details.

B Command Class Reference

Command Classes are groups of wireless commands that allow using certain functions of a Z-Wave device. In Z-Way each Z-Wave device has a data holder entry for each Command Class supported. During the inclusion and interview of the device the Command Class structure is instantiated in the data holder and filled with certain data. Command Class commands change values of the corresponding data holder structure. The follow list shows the public commands of the Command Classes supported with their parameters and the data holder objects changed.

In **Z-Wave Expert User Interface** navigate to `Configuration > Expert Commands` to execute commands of the supported Command Classes and visualizes all data holder elements in as tree in a simplified user interface.

B.1 Command Class Basic (0x20/32)

Version 1, Supported and Controlled

The Basic Command Class is the wildcard command class. Almost all Z-Wave devices support this command class but they interpret it's commands in different ways. A thermostat will handle a Basic Set Command in a different way than a Dimmer but both accept the Basic Set command and act. Used for generic interoperability between devices. You should always use more specific Command Classes where possible.

Data holders:

- **level**: Generic switching level of the device controlled

Command Basic Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_basic_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Basic Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

Command Basic Set

Syntax: Set(value, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_basic_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE value, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Basic Set

Parameter value: Value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

B.2 Command Class Wakeup (0x84/132)

Version 2, Controlled

Allows to manage periodical wakeup of sleeping battery operated device. Upon wakeup device will notify one node listed in nodeId. NB! If the device can wake up by interrupt (user interaction, button press, sensor trigger), it might happen that the device never wakes up. This can happen if you wake up the device by interrupe each time before internal chip wakeup period (usually from 1 to 4 minutes) reaches. (Z-Wave chip can not count for remaining time to next wakeup, so will restart timer again). This means that strictly speaking you can not rely on long time no wake up as an indicator of lost/damaged device or battery empty. NB! To save battery it is recommended to tune wakeup period to one week or even more for devices that do only need to report battery on wakeup (remote controls). For sensors it is recommended to have at least one hour wakeup period.

Data holders:

- **interval**: Wakeup interval in seconds
- **nodeId**: Node to notify about wakeup
- **min**: Minimal possible wakeup interval
- **max**: Maximal possible wakeup interval
- **default**: Factory default wakeup interval
- **step**: Step for wakeup interval (values are rounded to next or previous step)
- **lastWakeup**: Last time the device has sent us wake notification (Unix timestamp)

- **lastSleep**: Last time the device was sent into sleep mode (Unix timestamp)

Command Wakeup Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_wakeup_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Wakeup Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: interval and nodeId updated

Command Wakeup CapabilitiesGet

Syntax: CapabilitiesGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_wakeup_capabilities_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Wakeup CapabilityGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: min, max, default, step updated

Command Wakeup Sleep

Syntax: Sleep(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_wakeup_sleep(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Wakeup NoMoreInformation (Sleep)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: lastSleep updated

Command Wakeup Set

Syntax: Set(interval, notificationNodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_wakeup_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int interval, ZWBYTE notification_node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Wakeup Set

Parameter interval: Wakeup interval in seconds

Parameter notificationNodeId: Node Id to be notified about wakeup

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: interval and nodeId updated

B.3 Command Class NoOperation (0x00/0)

Used to check if device is reachable by sending empty packet.

B.4 Command Class Battery (0x80/128)

Version 1, Controlled

Allows monitoring the battery charging level of a device.

Data holders:

- **last**: Last battery level reported (0..100%)
- **lastChange**: Time (UNIX timestamp) when the battery was replaced last time (time of the moment when the value reported was way bigger than previous one)
- **history**: Subtree with history
- **[% value]**: Time when battery level reached this % value (0, 10, 20,... 100)

Command Battery Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_battery_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Battery Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: last updated. lastChange updated if battery level is way higher than it was before, history updated if reached next 10% step

B.5 Command Class ManufacturerSpecific (0x72/114)

Version 2, Supported and Controlled

Reports vendor information, product type and ID and device serial number.

Data holders:

- **vendorId**: Vendor ID assigned by Silicon Labs
- **vendor**: Vendor name
- **productId**: Product ID
- **productType**: Product Type ID
- **serialNumber**: Product Serial Number

Command ManufacturerSpecific Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_manufacturer_specific_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ManufacturerSpecific Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command ManufacturerSpecific DeviceIdGet

Syntax: DeviceIdGet(type, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_manufacturer_specific_device_id_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE type, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ManufacturerSpecific Device Id Get

Parameter type: Device Id type to request

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.6 Command Class Proprietary (0x88/136)

Version 1, Controlled

Allows to transfer manufacturer proprietary data. Data format is manufacturer specific.

Data holders:

- **bytes**: Binary bytes array of raw data

Command Proprietary Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_proprietary_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Proprietary Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Proprietary Set

Syntax: Set(data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_proprietary_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Proprietary Set

Parameter data: Data to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.7 Command Class Configuration (0x70/112)

Version 1, Controlled

Used to set certian configuration valeus that change the behavior of the device. Z-Wave requires that every device works out of the box without further configuration. However different configuration value significantly enhance the value a device. Z-Wave does not provide any information about the configuration values by wireless commands. User have to look into the device manual to learn about configuration parameters. The Device Description Record (ZDDX), incoprotated by Z-Way gives information about valid parameters and the meaning of the values to be set.

Data holders:

- **[paramId]**: Configuration parameter subtree.
- **val**: Value assigned
- **size**: Size of that parameter (1, 2 or 4 bytes)

Command Configuration Get

Syntax: Get(parameter, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_configuration_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE parameter, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Configuration Get

Parameter parameter: Parameter number (from 1 to 255)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: parameter subtree updated or created if absent

Command Configuration Set

Syntax: Set(parameter, value, size = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_configuration_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE parameter, int value, ZWBYTE size, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Configuration Set

Parameter parameter: Parameter number (from 1 to 255)

Parameter value: Value to be sent (negative and positive values are accepted, but will be stripped to size)

Parameter size: Size of the value (1, 2 or 4 bytes). Use 0 to guess from previously reported value if any. 0 means use size previously obtained Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: parameter subtree updated or created if absent

Command Configuration SetDefault

Syntax: SetDefault(parameter, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_configuration_set_default(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE parameter, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Configuration SetDefault

Parameter parameter: Parameter number to be set to device default

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: parameter subtree updated or created if absent

B.8 Command Class SensorBinary (0x30/48)

Version 2, Controlled

Allows receive binary sensor states.

Data holders:

- **typemask**: Internal. Bit mask of the supported types
- **[sensorType]**: Subtree for sensor type Id
- **sensorTypeString**: Description of sensor type
- **level**: Triggered/idle status

Command SensorBinary Get

Syntax: Get(sensorType = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sensor_binary_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int sensorType, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SensorBinary Get

Parameter sensorType: Type of sensor to query information for. 0xFF to query information for the first available sensor type. -1 to query information for all supported sensor types

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: sensorType subtree updated

B.9 Command Class Association (0x85/133)

Version 2, Supported and Controlled

Allows to manage the association groups: adding and removing nodeIDs in the association groups.

Data holders:

- **groups**: Number of association groups in the device
- **[groupId]**: Group subtree, where groupId = 1..groups
- **max**: Number of nodes the group can hold
- **nodes**: Array with nodes in the group
- **nodesToFollow**: Internal
- **specificGroup**: Number of specific association groups in the device

Command Association Get

Syntax: Get(groupId = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Association Get

Parameter groupId: Group Id (from 1 to 255). 0 requests all groups

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command Association Set

Syntax: Set(groupId, includeNode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZWBYTE include_node, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Association Set (Add)

Parameter groupId: Group Id (from 1 to 255)

Parameter includeNode: Node to be added to the group

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command Association Remove

Syntax: Remove(groupId, excludeNode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_remove(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZWBYTE exclude_node, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Association Remove

Parameter groupId: Group Id (from 1 to 255)

Parameter excludeNode: Node to be removed from the group

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command Association GroupingsGet

Syntax: GroupingsGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_groupings_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Association GroupingsGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Update number of supported groups and interview all groups

B.10 Command Class Meter (0x32/50)

Version 4, Controlled

Allows to read different kind of meters. Z-Wave differentiates different meter types and different meter scales. Please refer to the file translations/Scales.xml for details about possible meter types and values.

Data holders:

- **scalemask**: Internal. Bit mask with supported scales
- **resettable**: Flag to indicate of the meter can be resetted
- **[scaleId]**: Meter scale subtree
- **scale**: Meter scale id
- **scaleString**: Meter scale name
- **sensorType**: Sensor type id
- **sensorTypeString**: Sensor type name
- **val**: Meter value
- **ratetype**: Rate type
- **delta**: Delta from the last value requested
- **previous**: Previous value requested

Command Meter Get

Syntax: Get(scale = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int scale, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Meter Get

Parameter scale: Desired scale. -1 for all scales

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: scale subtree updated

Command Meter Reset

Syntax: Reset(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_reset(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Meter Reset

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: scale subtree updated

Command Meter Supported

Syntax: Supported(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_supported(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Meter SupportedGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.11 Command Class Meter Pulse (0x35/53)

Version 1, Controlled

Allows to gather information from pulse meters.

Data holders:

- **val**: Meter pulse value

Command MeterPulse Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_pulse_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MeterPulse Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.12 Command Class SensorMultilevel (0x31/49)

Version 11, Controlled

Allows to read different kind of sensor. Z-Wave differentiates different sensor types and different scales of this sensor. Please refer to the file /translations/scales.xml for details about possible sensor types and values.

Data holders:

- **typemask**: Internal. Bit mask of the supported types
- **[sensorTypeId]**: Subtree for sensor type Id
- **sensorTypeString**: Description of sensor type
- **scale**: Scale Id
- **scaleString**: Scale description
- **val**: Value

- **size**: Internal. Size of the value (1, 2 or 4 bytes)
- **precision**: Internal. Precision used in value (number of digits after decimal dot)
- **deviceScale**: Internal. Scale Id on the device's side (if local conversion is used, like C->F)

Command SensorMultilevel Get

Syntax: Get(sensorType = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sensor_multilevel_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int sensor_type, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SensorMultilevel Get

Parameter sensorType: Type of sensor to be requested. -1 means all sensor types supported by the device

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: sensorTypeId subtree updated

B.13 Command Class Sensor Configuration (0x9E/158)

Version 1, Controlled

Allows to configure senors specific configuration like trigger level. Works in conjunction with SensorMultilevel Command Class. In modern devices replaced by Configuration Command Class.

Data holders:

- **sensorType**: Sensor type Id
- **sensorTypeString**: Sensor type descirption
- **val**: Trigger value
- **scale**: Scale of trigger value
- **scaleString**: Scale description
- **size**: Internal. Size of the value (1, 2 or 4 bytes)
- **precision**: Internal. Precision used in value (number of digits after decimal dot)

Command SensorConfiguration Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sensor_configuration_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SensorConfiguration Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: all dataholders are updated

Command SensorConfiguration Set

Syntax: Set(mode, value, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sensor_configuration_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE mode, float value, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SensorConfiguration Set

Parameter mode: Value set mode

Parameter value: Value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: all dataholders are updated

B.14 Command Class SwitchAll (0x27/39)

Version 1, Supported and Controlled

Controls the behavior of a actuator on Switch All commands. Also allows to send Switch All commands.

Data holders:

- **mode**: Which type of SwitchAll On/Off commands to react on: 0 for none, 1 to reacto on Off only, 2 to react on On only, 255 to react on both
- **onOff**: Allows to trigger SwitchAll On/Off commands from other devices. Set to False on Off command received and True on On command.

Command SwitchAll Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_all_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchAll Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode updated

Command SwitchAll Set

Syntax: Set(mode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_all_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE mode, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchAll Set

Parameter mode: SwitchAll Mode: see definitions below

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode updated

Command SwitchAll SetOn

Syntax: SetOn(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_all_set_on(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchAll Set On

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SwitchAll SetOff

Syntax: SetOff(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_all_set_off(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchAll Set Off

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.15 Command Class SwitchBinary (0x25/37)

Version 1, Supported and Controlled

Allows to control On/Off switches, actuators, electrical power switches and trap On/Off control commands from other devices.

Data holders:

- **level**: State: False for Off, True for On

Command SwitchBinary Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_binary_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchBinary Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

Command SwitchBinary Set

Syntax: Set(value, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_binary_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBOOL value, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchBinary Set

Parameter value: Value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

B.16 Command Class SwitchMultilevel (0x26/38)

Version 1, Supported and Controlled

Allows to control all actuators with multilevel switching functions, primarily Dimmers and Motor Controlling devices as well as trap dim events sent by remotes.

Data holders:

- **level**: State 0...99 = 0...100%, 255 for On on last value (or on maximum - device specific)
- **startChange**: Dimming up or down. Updated on dimming start. Allows to trap events from remotes to controller.
- **stopChange**: Updated on dimming end. Allows to trap events from remotes to controller.
- **prevLevel**: Internal
- **primary**: Unused
- **secondary**: Unsued

Command SwitchMultilevel Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_multilevel_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchMultilevel Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

Command SwitchMultilevel Set

Syntax: Set(level, duration = 0xff, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_multilevel_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE level, ZWBYTE duration, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchMultilevel Set

Parameter level: Level to be set

Parameter duration: Duration of change:. 0 instantly. 0x01...0x7f in seconds. 0x80...0xfe in minutes mapped to 1...127 (value 0x80=128 is 1 minute). 0xff use device factory default

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

Command SwitchMultilevel StartLevelChange

Syntax: StartLevelChange(dir, duration = 0xff, ignoreStartLevel = TRUE, startLevel = 50, incdec = 0, step = 0xff, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_multilevel_start_level_change(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE dir, ZWBYTE duration, ZWBOOL ignoreStartLevel, ZWBYTE startLevel, ZWBYTE incdec, ZWBYTE step, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchMultilevel StartLevelChange

Parameter dir: Direction of change: 0 to incrase, 1 to decrase

Parameter duration: Duration of change:. 0 instantly. 0x01...0x7f in seconds. 0x80...0xfe in minutes mapped to 1...127 (value 0x80=128 is 1 minute). 0xff use device factory default

Parameter ignoreStartLevel: If set to True, device will ignore start level value and will use it's curent value

Parameter startLevel: Start level to change from

Parameter incdec: Increment/decrement type for step:. 0 Increment. 1 Decrement. 2 Reserved. 3 No Inc/Dec

Parameter step: Step to be used in level change in percentage. 0...99 mapped to 1...100%. 0xff uses device factory default

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

Command SwitchMultilevel StopLevelChange

Syntax: StopLevelChange(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_multilevel_stop_level_change(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchMultilevel StopLevelChange

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level updated

B.17 Command Class MultiChannelAssociation (0x8E/142)

Version 3, Supported and Controlled

This is an extention to the Association Command Class. It follows the same logic as the Association Command Class and has the same commands but accepts different instance values.

Data holders:

- **groups**: Number of association groups in the device (can be smaller than the number of groups in Association)
- **[groupId]**: Group subtree, where groupId = 1..groups
- **max**: Number of nodes/instances the group can hold
- **nodesInstances**: Array with nodes/instances in the group. Each pair is represented by two elements (node, instance).
- **nodesInstancesToFollow**: Internal

Command MultiChannelAssociation Get

Syntax: Get(groupId = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_association_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannelAssociation Get

Parameter groupId: Group Id (from 1 to 255). 0 requests all groups

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command MultiChannelAssociation Set

Syntax: Set(groupId, includeNode, includeInstance, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_association_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZWBYTE include_node, ZWBYTE include_instance, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannelAssociation Set (Add)

Parameter groupId: Group Id (from 1 to 255)

Parameter includeNode: Node to be added to the group

Parameter includeInstance: Instance of the node to be added to the group

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command MultiChannelAssociation Remove

Syntax: Remove(groupId, excludeNode, excludeInstance, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_association_remove(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group_id, ZWBYTE exclude_node, ZWBYTE exclude_instance, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannelAssociation Remove

Parameter groupId: Group Id (from 1 to 255)

Parameter excludeNode: Node to be removed from the group

Parameter excludeInstance: Instance of the node to be removed from the group

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Subtree corresponding to the group updated

Command MultiChannelAssociation GroupingsGet

Syntax: GroupingsGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_association_groupings_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannelAssociation GroupingsGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.18 Command Class MultiChannel (0x60/96)

Version 4, Supported and Controlled

Allows to communicate with internal parts of device called channels or instances. Implemented transparently by the library.

Data holders:

- **endPoints**: Number of endpoints
- **[endPointId]**: Endpoint ID
- **aggregated**: Number of aggregated endpoints
- **[endPointId]**: Aggregated endpoint ID (numbering starts from endPoints + 1)
- **dynamic**: Flag describing if endpoins are dynamic (their number and type can change over time)
- **identical**: Internal. Flag describing if endpoins are identical
- **myInstance**: Internal
- **doneIds**: Internal

Command MultiChannel Get

Syntax: Get(ccId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE cc_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannel Get (MultiInstance V1 command). Reports number of channels supporting a defined Command Class. Depricated by MutliChannel V2, needed for old devices only

Parameter ccId: Command Class Id in question

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MultiChannel EndpointFind

Syntax: EndpointFind(generic, specific, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_endpoint_find(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE generic, ZWBYTE specific, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannel Endpoint Find. Note that MultiChannel EndpointFind Report is not supported as useless. But one can still trap the response packet in logs

Parameter generic: Generic type in search

Parameter specific: Specific type in search

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MultiChannel EndpointGet

Syntax: EndpointGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_endpoint_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannel Endpoint Get. Get the number of available endpoints

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MultiChannel CapabilitiesGet

Syntax: CapabilitiesGet(endpoint, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_capabilities_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE endpoint, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannel Capabilities Get. Request information about the specified endpoint

Parameter endpoint: Endpoint in question

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MultiChannel AggregatedMembersGet

Syntax: AggregatedMembersGet(endpoint, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_multichannel_aggregated_members_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE endpoint, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send MultiChannel Aggregated Members Get. Request information about endpoints in the specified aggregated endpoint (v4 and above)

Parameter endpoint: Aggregated endpoint in question

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.19 Command Class Node Naming (0x77/119)

Version 1, Supported and Controlled

Allows assigning a readable string for a name and a location to a physical device. The two strings are stored inside the device and can be obtained upon request. There are no restrictions to the name except the maximum length up

to 16 characters.

Data holders:

- **nodename**: Node name
- **nameEncoding**: NodeName encoding
- **location**: Location
- **locationEncoding**: Location encoding

Command NodeNaming Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_node_naming_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NodeNaming GetName and GetLocation

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: nodename, nameEncoding, location and locationEncoding updated

Command NodeNaming GetName

Syntax: GetName(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_node_naming_get_name(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NodeNaming GetName

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: nodename and nameEncoding updated

Command NodeNaming GetLocation

Syntax: GetLocation(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_node_naming_get_location(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NodeNaming GetLocation

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: location and locationEncoding updated

Command NodeNaming SetName

Syntax: SetName(name, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_node_naming_set_name(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWCSTR name, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NodeNaming SetName

Parameter name: Value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: nodename and nameEncoding updated

Command NodeNaming SetLocation

Syntax: SetLocation(location, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_node_naming_set_location(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWCSTR location, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NodeNaming SetLocation

Parameter location: Value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: location and locationEncoding updated

B.20 Command Class Thermostat SetPoint (0x43/67)

Version 3, Controlled

Allows to set a certain setpoint to a thermostat (set temperature to maintain). The command class can be applied to different kind of thermostats (heating, cooling, ...), hence it has various modes.

Data holders:

- **[modeId]**: Subtree for mode
- **modeName**: Mode description
- **scale**: Scale Id
- **scaleString**: Scale description
- **val**: Temperature to maintain
- **setVal**: Last set temperature to maintain (might differ from val until thermostat wakeup)
- **min**: Minimal temperature value supported by the device
- **max**: Maximal temperature value supported by the device
- **size**: Internal. Size of the value (1, 2 or 4 bytes)
- **precision**: Internal. Precision used in value (number of digits after decimal dot)
- **deviceScale**: Internal. Scale Id on the device side (if local conversion is used, like C->F)
- **deviceScaleString**: Internal. Scale description of the device
- **modemask**: Internal. Bit mask with supported modes
- **danfossBugFlag**: Internal

Command ThermostatSetPoint Get

Syntax: Get(mode = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_setpoint_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int mode, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatSetPoint Get

Parameter mode: Thermostat Mode. -1 requests for all modes

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: modeId subtree updated

Command ThermostatSetPoint Set

Syntax: Set(mode, value, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_setpoint_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int mode, float value, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatSetPoint Set

Parameter mode: Thermostat Mode

Parameter value: temperature

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: modeId subtree updated

B.21 Command Class Thermostat Mode (0x40/64)

Version 3, Controlled

Allows to switch a heating/cooling actuator in different modes.

Data holders:

- **modemask**: Internal. Bit mask with supported modes
- **mode**: Current mode
- **[modeId]**: Mode subtree
- **modeName**: Mode description

Command ThermostatMode Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_mode_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatMode Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command ThermostatMode Set

Syntax: Set(mode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_mode_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE mode, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatMode Set

Parameter mode: Thermostat Mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.22 Command Class Thermostat Fan Mode (0x44/68)

Version 4, Controlled

Allows to controls fan modes in thermostats.

Data holders:

- **modemask**: Internal. Bit mask with supported modes
- **mode**: Current mode
- **[modeId]**: Mode subtree
- **modeName**: Mode description
- **on**: Reports if fan is currently On (True) or Off (False)

Command ThermostatFanMode Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_fan_mode_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatFanMode Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode and on updated

Command ThermostatFanMode Set

Syntax: Set(on, mode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_fan_mode_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBOOL on, ZWBYTE mode, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatFanMode Set

Parameter on: TRUE to turn fan on (and set mode), FALSE to comletely turn off (mode is ignored)

Parameter mode: Thermostat Fan Mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode and on updated

B.23 Command Class Thermostat Fan State (0x45/69)

Version 2, Controlled

Allows to determine the operating state of the fan. V2 is not yet implemented.

Data holders:

- **state**: Fan current state (0 Off, 1 Running)

Command ThermostatFanState Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_fan_state_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatFanState Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: state and on updated

B.24 Command Class Thermostat Operating State (0x42/66)

Version 2, Controlled

Allows to determine the operating state of the thermostat and state change history.

Data holders:

- **state**: Current operation state
- **statemask**: Internal. Bit mask of supported logs for each state

- **[stateId]**: Subtree with state log info
- **today**: Number of minutes thermostat was in this state today (since 0:00)
- **yesterday**: Number of minutes thermostat was in this state yesterday (since 0:00)

Command ThermostatOperatingState Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_operating_state_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatOperatingState Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: state updated

Command ThermostatOperatingState LoggingGet

Syntax: LoggingGet(state, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_thermostat_operating_state_logging_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE state, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ThermostatOperatingState Logging Get

Parameter state: State number to get logging for. 0 to get log for all supported states

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: stateId subtree updated updated

B.25 Command Class Alarm Sensor (0x9C/156)

Version 1, Controlled

Deprecated Command Class. Now Alarm/Notification is used instead.

Data holders:

- **alarmMap**: Internal. Bit mask of supported alarm types
- **alarms**: Unused
- **[alarmTypeId]**: Alarm type subtree
- **srcId**: Source of event
- **sensorState**: Alarm state
- **sensorTime**: Alarm time (according to the sender)
- **typeString**: Name of alarm type

Command AlarmSensor SupportedGet

Syntax: SupportedGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_alarm_sensor_supported_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send AlarmSensor SupportedGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: List of supported types updated

Command AlarmSensor Get

Syntax: Get(type = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_alarm_sensor_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int type, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send AlarmSensor Get. Requests the status of the alarm sensor of a given type

Parameter type: Alarm type to get. -1 means get all types

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Alarm type subtree updated

B.26 Command Class Door Lock (0x62/98)

Version 2, Controlled

Allows to operate an electronic door lock. This Command Class is ALWAYS encapsulated in Security. Door lock modes are the following:.
- 0x00 Door Unsecured (Open).
- 0x01 Door Unsecured with timeout.
- 0x10 Door Unsecured for inside Door Handles.
- 0x11 Door Unsecured for inside Door Handles with timeout.
- 0x20 Door Unsecured for outside Door Handles.
- 0x21 Door Unsecured for outside Door Handles with timeout.
- 0xFE Door/Lock Mode Unknown (bolt not fully retracted/engaged).
- 0xFF Door Secured (closed).

Data holders:

- **mode**: Operating mode of the lock
- **insideMode**: Bit mask describing if a specific handles (1..4) can open the door from inside
- **outsideMode**: Bit mask describing if a specific handles (1..4) can open the door from outside
- **lockMinutes**: Time remaind before autolock (minutes, 0xFE for no autolock)
- **lockSeconds**: Time remaind before autolock (seconds, 0xFE for no autolock)
- **condition**: Bit mask describing lock components: bit 0: Door Open(0)/Close(1), bit 1: Bolt Locked(0)/Unlocked(1), bit 2: Latch Open(0)/Close(1)
- **insideState**: Bit mask describing if a specific handles (1..4) can open the door from inside
- **outsideState**: Bit mask describing if a specific handles (1..4) can open the door from outside
- **timeoutMinutes**: Timeout for autolock (minutes, 0xFE for no autolock)
- **timeoutSeconds**: Timeout for autolock (seconds, 0xFE for no autolock)
- **opType**: 0x01 for constant operation, 0x02 for autolock

Command DoorLock Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_door_lock_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send DoorLock Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode, insideMode, outsideMode, lockMinutes, lockSeconds and condition updated

Command DoorLock ConfigurationGet

Syntax: ConfigurationGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_door_lock_configuration_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send DoorLock Configuration Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: insideState, outsideState, timeoutMinutes, timeoutSeconds, opType updated

Command DoorLock Set

Syntax: Set(mode, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_door_lock_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE mode, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send DoorLock Set

Parameter mode: Lock mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: mode, insideMode, outsideMode, lockMinutes, lockSeconds and condition updated

Command DoorLock ConfigurationSet

Syntax: ConfigurationSet(opType, outsideState, insideState, lockMin, lockSec, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_door_lock_configuration_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE opType, ZWBYTE outsideState, ZWBYTE insideState, ZWBYTE lockMin, ZWBYTE lockSec, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send DoorLock Configuration Set

Parameter opType: Operation type

Parameter outsideState: State of outside door handle

Parameter insideState: State of inside door handle

Parameter lockMin: Lock after a specified time (minutes part)

Parameter lockSec: Lock after a specified time (seconds part)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: insideState, outsideState, timeoutMinutes, timeoutSeconds, opType updated

B.27 Command Class Door Lock Logging (0x4C/76)

Version 1, Controlled

Allows to receive reports about all successful and failed activities of the electronic door lock. Event types are the following:.
- 1 Lock Command: Keypad access code verified lock command.
- 2 Unlock Command: Keypad access code verified unlock command.
- 3 Lock Command: Keypad lock button pressed.
- 4 Unlock command: Keypad unlock button pressed.
- 5 Lock Command: Keypad access code out of schedule.
- 6 Unlock Command: Keypad access code out of schedule.
- 7 Keypad illegal access code entered.
- 8 Key or latch operation locked (manual).

- 9 Key or latch operation unlocked (manual).
- 10 Auto lock operation.
- 11 Auto unlock operation.
- 12 Lock Command: Z-Wave access code verified.
- 13 Unlock Command: Z-Wave access code verified.
- 14 Lock Command: Z-Wave (no code).
- 15 Unlock Command: Z-Wave (no code).
- 16 Lock Command: Z-Wave access code out of schedule.
- 17 Unlock Command Z-Wave access code out of schedule.
- 18 Z-Wave illegal access code entered.
- 19 Key or latch operation locked (manual).
- 20 Key or latch operation unlocked (manual).
- 21 Lock secured.
- 22 Lock unsecured.
- 23 User code added.
- 24 User code deleted.
- 25 All user codes deleted.
- 26 Master code changed.
- 27 User code changed.
- 28 Lock reset.
- 29 Configuration changed.
- 30 Low battery.
- 31 New Battery installed.

Data holders:

- **maxRecords**: Maximum number of records the lock can store. Olded records are reused first.
- **[recordId]**: Subtree storing log record
- **time**: Time of the event
- **event**: Event type
- **uId**: UserID (from UserCode Command Class)
- **eventString**: Event type description

Command DoorLockLogging Get

Syntax: Get(record = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_door_lock_logging_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE record, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send DoorLockLogging Get

Parameter record: Record number to get, or 0 to get last records

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: record subtree updated

B.28 Command Class User Code (0x63/99)

Version 1, Controlled

Allows to define individual user entry code in electrnic door lock.

Data holders:

- **maxUsers**: Maximum number of supported users
- **[userId]**: User subtree
- **code**: User code

- **status**: Status of the user: 0 for available (no code set), 1 for occupied (code set), 2 for reserved by administrator
- **hasCode**: Flag if a valid code is set (in case device reports occupied, but code is not valid (less than 4 symbols) or code not set but old is still reported by the device)

Command UserCode Get

Syntax: Get(user = -1, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_user_code_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send UserCode Get

Parameter user: User index to get code for (1...maxUsers). -1 to get codes for all users

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId subtree updated

Command UserCode Set

Syntax: Set(user, code, status, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_user_code_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWCSTR code, ZWBYTE status, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send UserCode Set

Parameter user: User index to set code for (1...maxUsers). 0 means set for all users

Parameter code: Code to set (4...10 characters long)

Parameter status: Code status to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId subtree updated

Command UserCode SetRaw

Syntax: SetRaw(user, code, status, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_user_code_set_raw(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, const ZWBYTE * code, ZWBYTE status, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send UserCode Set (raw)

Parameter user: User index to set code for (1...maxUsers). 0 means set for all users

Parameter code: Code to set (4...10 bytes long)

Parameter status: Code status to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId subtree updated

B.29 Command Class Time (0x8A/138)

Version 2, Supported and Controlled

Allows to report to devices in Z-Wave network time and date as well as time zone offset and daylight savings parameters. The data formats are based on the International Standard ISO 8601.

Command Time TimeGet

Syntax: TimeGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_time_time_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Time TimeGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Time DateGet

Syntax: DateGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_time_date_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Time DateGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Time OffsetGet

Syntax: OffsetGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_time_offset_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Time TimeOffsetGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.30 Command Class Time Parameters (0x8B/139)

Version 1, Controlled

Used to set date and time. Time zone offset and daylight savings may be set in the Time Command Class. The data formats are based on the International Standard ISO 8601.

Command TimeParameters Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_time_parameters_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send TimeParameters Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command TimeParameters Set

Syntax: Set(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_time_parameters_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send TimeParameters Set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.31 Command Class Clock (0x81/129)

Version 1, Supported and Controlled

Sync clock on the device with controller system clock.

Command Clock Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_clock_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Clock Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: reported value ignored

Command Clock Set

Syntax: Set(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_clock_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Clock Set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.32 Command Class Scene Activation (0x2B/43)

Version 1, Supported and Controlled

Allows to activate scenes on devices and trap scene activation events from remotes.

Data holders:

- **currentScene**: Scene activated from remote
- **dimmingDuration**: Dimming duration for the activated scene

Command SceneActivation Set

Syntax: Set(sceneId, dimmingDuration = 0xff, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_scene_activation_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE sceneId, ZWBYTE dimmingDuration, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SceneActivation Set

Parameter sceneId: Scene Id

Parameter dimmingDuration: Dimming duration

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.33 Command Class Scene Controller Conf (0x2D/45)

Version 1, Controlled

Allows to set scene Id to be activated using SceneActivation Command Class on a remote.

Data holders:

- **[groupId]**: Subtree for a given association group number (defined by Association Command Class)
- **scene**: Scene to activate for all devices in the group
- **duration**: Duration for scene activation

Command SceneControllerConf Get

Syntax: Get(group = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_scene_controller_conf_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SceneControllerConf Get

Parameter group: Group Id. 0 requests all groups

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: group subtree updated

Command SceneControllerConf Set

Syntax: Set(group, scene, duration = 0x0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_scene_controller_conf_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE group, ZWBYTE scene, ZWBYTE duration, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SceneControllerConf Set

Parameter group: Group Id

Parameter scene: Scene Id

Parameter duration: Duration

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: group subtree updated

B.34 Command Class Scene Actuator Conf (0x2C/44)

Version 1, Controlled

Allows to configure actuators to set specified level on a given scene activation by SceneActivation Command Class.

Data holders:

- **[sceneId]**: Subtree for scene
- **level**: Level to set on scene activation
- **dimming**: Default dimming duration to use
- **currentScene**: Currently activated scene

Command SceneActuatorConf Get

Syntax: Get(scene = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_scene_actuator_conf_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE scene, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SceneActuatorConf Get

Parameter scene: Scene Id. 0 means get current scene

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: scene subtree updated, currentScene updated (if scene = 0)

Command SceneActuatorConf Set

Syntax: Set(scene, level, dimming = 0xff, override = TRUE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_scene_actuator_conf_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE scene, ZWBYTE level, ZWBYTE dimming, ZWBOOL override, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SceneActuatorConf Set

Parameter scene: Scene Id

Parameter level: Level

Parameter dimming: Dimming

Parameter override: If false then the current settings in the device is associated with the Scene Id. If true then the Level value is used

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: scene subtree updated

B.35 Command Class Indicator (0x87/135)

Version 1, Controlled

Operates the indicator on the device if available. Can be used to identify a device or use the indicator for special purposes (show away/at home mode).

Data holders:

- **stat**: Status of the indicator

Command Indicator Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_indicator_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Indicator Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: stat updated

Command Indicator Set

Syntax: Set(val, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_indicator_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE val, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Indicator Set

Parameter val: Value to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: stat updated

B.36 Command Class Protection (0x75/117)

Version 2, Controlled

Allows to disable local and RF control of the device.

Data holders:

- **state**: Local control state (0 = Unprotected, 1 = Protected by sequence, 2 = Protected)
- **rfState**: Control via RF state (0 = Unprotected, 1 = No RF control, 2 = No RF response at all)
- **exclusive**: Flag describing if exclusive control via RF is supported
- **timeout**: Flag describing if timeout of protection of control via RF is supported
- **stateCap**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support
- **rfStateCap**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support
- **exclusiveCap**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support
- **timeoutCap**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support

Command Protection Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: state, rfState updated

Command Protection Set

Syntax: Set(state, rfState = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE state, ZWBYTE rfState, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Set

Parameter state: Local control protection state

Parameter rfState: RF control protection state

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: state and rfState updated

Command Protection ExclusiveGet

Syntax: ExclusiveGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_exclusive_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Exclusive Control Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Protection ExclusiveSet

Syntax: ExclusiveSet(controlNodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_exclusive_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE controlNodeId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Exclusive Control Set

Parameter controlNodeId: Node Id to have exclusive control over destination node

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Protection TimeoutGet

Syntax: TimeoutGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_timeout_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Timeout Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command Protection TimeoutSet

Syntax: TimeoutSet(timeout, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_protection_timeout_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int timeout, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Protection Timeout Set

Parameter timeout: Timeout in seconds. 0 is no timer set. -1 is infinite timeout. max value is 191 minute (11460 seconds). values above 1 minute are rounded to 1 minute boundary

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.37 Command Class Schedule Entry Lock (0x4E/78)

Version 3, Controlled

Allows to define individual time intervals for access to a door lock per user. Refers to users defined by User Code Command Class.

Data holders:

- **weekDaySlots**: Number of weekday slots supported
- **yearSlots**: Number of date slots supported
- **[userId]**: Subtree for userId
- **Weekday**: Subtree for weekday schedule
- **[slotId]**: Subtree slotId
- **dayOfWeek**: Day of week
- **startHour**: Start hour
- **startMinute**: Start minute

- **stopHour**: Stop hour
- **stopMinute**: Stop minute
- **Year**: Subtree for date schedule
- **[slotId]**: Subtree slotId
- **startYear**: Start year
- **startMonth**: Start month
- **startDay**: Start day
- **startHour**: Start hour
- **startMinute**: Start minute
- **stopYear**: Stop year
- **stopMonth**: Stop month
- **stopDay**: Stop day
- **stopHour**: Stop hour
- **stopMinute**: Stop minute

Command ScheduleEntryLock Enable

Syntax: Enable(user, enable, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_schedule_entry_lock_enable(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWBOOL enable, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ScheduleEntryLock Enable(All)

Parameter user: User to enable/disable schedule for. 0 to enable/disable for all users

Parameter enable: TRUE to enable schedule, FALSE otherwise

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command ScheduleEntryLock WeekdayGet

Syntax: WeekdayGet(user, slot, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_schedule_entry_lock_weekday_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWBYTE slot, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ScheduleEntryLock Weekday Get

Parameter user: User to get schedule for. 0 to get for all users

Parameter slot: Slot to get schedule for. 0 to get for all slots

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId->Weekday->slotId subtree updated

Command ScheduleEntryLock WeekdaySet

Syntax: WeekdaySet(user, slot, dayOfWeek, startHour, startMinute, stopHour, stopMinute, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_schedule_entry_lock_weekday_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWBYTE slot, ZWBYTE dayOfWeek, ZWBYTE startHour, ZWBYTE startMinute, ZWBYTE stopHour, ZWBYTE stopMinute, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ScheduleEntryLock Weekday Set

Parameter user: User to set schedule for

Parameter slot: Slot to set schedule for

Parameter dayOfWeek: Weekday number (0..6). 0 = Sunday. . 6 = Saturday

Parameter startHour: Hour when schedule starts (0..23)

Parameter startMinute: Minute when schedule starts (0..59)

Parameter stopHour: Hour when schedule stops (0..23)

Parameter stopMinute: Minute when schedule stops (0..59)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId->Weekday->slotId subtree updated

Command ScheduleEntryLock YearGet

Syntax: YearGet(user, slot, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_schedule_entry_lock_year_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWBYTE slot, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ScheduleEntryLock Year Get

Parameter user: User to enable/disable schedule for. 0 to get for all users

Parameter slot: Slot to get schedule for. 0 to get for all slots

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId->Year->slotId subtree updated

Command ScheduleEntryLock YearSet

Syntax: YearSet(user, slot, startYear, startMonth, startDay, startHour, startMinute, stopYear, stopMonth, stopDay, stopHour, stopMinute, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_schedule_entry_lock_year_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int user, ZWBYTE slot, ZWBYTE startYear, ZWBYTE startMonth, ZWBYTE startDay, ZWBYTE startHour, ZWBYTE startMinute, ZWBYTE stopYear, ZWBYTE stopMonth, ZWBYTE stopDay, ZWBYTE stopHour, ZWBYTE stopMinute, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ScheduleEntryLock Year Set

Parameter user: User to set schedule for

Parameter slot: Slot to set schedule for

Parameter startYear: Year in current century when schedule starts (0..99)

Parameter startMonth: Month when schedule starts (1..12)

Parameter startDay: Day when schedule starts (1..31)

Parameter startHour: Hour when schedule starts (0..23)

Parameter startMinute: Minute when schedule starts (0..59)

Parameter stopYear: Year in current century when schedule stops (0..99)

Parameter stopMonth: Month when schedule stops (1..12)

Parameter stopDay: Day when schedule stops (1..31)

Parameter stopHour: Hour when schedule stops (0..23)

Parameter stopMinute: Minute when schedule stops (0..59)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: userId->Year->slotId subtree updated

B.38 Command Class Climate Control Schedule (0x46/70)

Version 1, Supported and Controlled

Obsolete but still partly implemented for legacy support.

Data holders:

- **overrideType**: Type of current override
- **overrideState**: State of override

Command ClimateControlSchedule OverrideGet

Syntax: OverrideGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_climate_control_schedule_override_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ClimateControlSchedule Override Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command ClimateControlSchedule OverrideSet

Syntax: OverrideSet(overrideType, overrideState, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_climate_control_schedule_override_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE overrideType, ZWBYTE overrideState, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ClimateControlSchedule Override Set

Parameter overrideType: Override type to set. (0 – no override, 1 – temporary override, 2 – permanent override)

Parameter overrideState: Override state. -128 (0x80) ... -1 (0xFF): setpoint -12.8 ... -0.1 degrees. 0 (0x00): setpoint. 1 (0x01) ... 120 (0x78): setpoint +0.1 ... +12 degrees. 121 (0x79): frost protection. 122 (0x7A): energy saving. 123 (0x7B) ... 126 (0x7D): reserved. 127 (0x7F): unused

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.39 Command Class MeterTableMonitor (0x3D/61)

Version 2, Controlled

Allows to read historical and accumulated values in physical units from a water meter or other metering device (gas, electric etc.) and thereby enabling automatic meter reading capabilities.

Data holders:

- **adminId**: Meter administrator ID
- **Id**: Customer ID
- **rateType**: Type of rate (export or import)
- **payMeter**: Specifies the way the account is done
- **meterType**: Meter type
- **meterTypeString**: Meter description
- **dataSetMask**: Internal. Bit mask with type of data set supported
- **dataSetHistoryMask**: Internal. Bit mask with type of data set history supported
- **maxHistory**: Max number of records the device can store
- **statusMask**: Internal. Bit mask with type of events supported
- **maxEvents**: Max number of events the device can store
- **[dataSetId]**: Subtree for data set
- **val**: Meter value for this data set
- **time**: Time corresponding to the value
- **scale**: Scale ID

- **scaleString**: Scale desctiption
- **history**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support
- **status**: Subtree with statuses
- **[statuseId]**: Subtree with specific status ID
- **statusString**: Status descirption
- **active**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support
- **time**: Requires Z-Wave specification re-read. Please contact Z-Wave.Me support

Command MeterTableMonitor StatusDateGet

Syntax: StatusDateGet(maxResults = 0, startDate, endDate, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_table_monitor_status_date_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE maxResults, time_t startDate, time_t endDate, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send StatusTableMonitor Status Get for a range of dates

Parameter maxResults: Maximum number of entries to get from log. 0 means all matching entries

Parameter startDate: Start date and time (local UNIX time)

Parameter endDate: End date and time (local UNIX time)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MeterTableMonitor StatusDepthGet

Syntax: StatusDepthGet(maxResults = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_table_monitor_status_depth_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE maxResults, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send StatusTableMonitor Status Get for specified depth

Parameter maxResults: Number of entries to get from log. 0 means current status only. 0xFF means all entries

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MeterTableMonitor CurrentDataGet

Syntax: CurrentDataGet(setId = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_table_monitor_current_data_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE setId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send StatusTableMonitor Current Data Get

Parameter setId: Index of dataset to get data for. 0 to get data for all supported datasets

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command MeterTableMonitor HistoricalDataGet

Syntax: HistoricalDataGet(setId = 0, maxResults = 0, startDate, endDate, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_meter_table_monitor_historical_data_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE setId, ZWBYTE maxResults, time_t startDate, time_t endDate, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send StatusTableMonitor Historical Data Get

Parameter setId: Index of dataset to get data for. 0 to get data for all supported datasets

Parameter maxResults: Maximum number of entries to get from log. 0 means all matching entries

Parameter startDate: Start date and time (local UNIX time)

Parameter endDate: End date and time (local UNIX time)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.40 Command Class Alarm (0x71/113)

Version 5, Controlled

Also known as Notification Command Class. Used to report alarm events from binary sensors. Starting from version 3 all types are strictly defines:.
- 0x01 Smoke.
- 0x02 CO.
- 0x03 CO2.
- 0x04 Heat.
- 0x05 Water.
- 0x06 Access Control.
- 0x07 Burglar.
- 0x08 Power Management.
- 0x09 System.
- 0x0a Emergency.
- 0x0b Clock.

Data holders:
- **V1supported**: boolean flag saying if version 1 (deprecated) is supported
- **V1event** : structure to store V1 events
- **alarmType**: V1 alarm type
- **level**: V1 status
- **typeMask**: bit mask of supported alarm types
- **[typeId]**: subtree to store events of specific alarm types
- **typeString**: name of the alarm type
- **status**: flag with alarm status (alarm enabled/disabled)
- **eventMask**: bit mask of supported events of this alarm type
- **event**: last event ID
- **eventString**: last event name
- **eventParameters**: last event parameters
- **eventSequence**: internal

Command Alarm Get

Syntax: Get(type = 0, event = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_alarm_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE type, ZWBYTE event, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Alarm Get. Requests the status of a specific event of a specific alarm type

Parameter type: Type of alarm to get level for. 0 to get level for all supported alarms (v2 and higher). 0xFF to get level for first supported alarm (v2 and higher)

Parameter event: Notification event to get level for. This argument is ignored prior to Notification v3. Must be 0 if type is 0xFF

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Alarm type subtree updated

Command Alarm Set

Syntax: Set(type, level, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_alarm_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE type, ZWBYTE level, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Alarm Set (v2 and higher). Enable/disable alarms of a specific type

Parameter type: Type of alarm to set level for

Parameter level: Level to set (0x0 = off, 0xFF = on, other values are reserved)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: Alarm type subtree updated

B.41 Command Class PowerLevel (0x73/115)

Version 1, Supported and Controlled

Used to set device power level and to test the link to a other devices in the network.

Data holders:

- **level**: Current power level (0 for normal power, 1..9 for -1..-9 dBm)
- **timeout**: Timeout of the power level set (after timeout the device turns back to normal power)
- **[nodeId]**: Subtree with report of a test with nodeId
- **status**: Current test status (0 = Failed, 1 = Successfully finished, 2 = In progress)
- **totalFrames**: Total frames sent
- **acknowledgedFrames**: Acknowledged frames from total sent

Command PowerLevel Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_power_level_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send PowerLevel Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level and timeout updated

Command PowerLevel Set

Syntax: Set(level, timeout, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_power_level_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE level, ZWBYTE timeout, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send PowerLevel Set

Parameter level: Power level to set (from 0 to 9)

Parameter timeout: Timeout in seconds

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: level and timeout updated

Command PowerLevel TestNodeGet

Syntax: TestNodeGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_power_level_test_node_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send PowerLevel Test Node Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: subtree with report for the given node updated

Command PowerLevel TestNodeSet

Syntax: TestNodeSet(testNodeId, level, frameCount, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_power_level_test_node_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE testNodeId, ZWBYTE level, int frameCount, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send PowerLevel Test Node Set. Starts sending specified number of NOP packets to a given device at a given power level. Once finished, unsolicited report MIGHT be sent by the device (at any time you can use TestNodeGet)

Parameter testNodeId: Node to set test packets to

Parameter level: Power level to use (from 0 to 9)

Parameter frameCount: Number of test frames to send (from 1 to 65535)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: subtree with report for the given node updated

B.42 Command Class Z-Wave Plus Info (0x5E/94)

Version 2, Supported and Controlled

Describes device Z-Wave Plus role and type.

Data holders:

- **plusVersion**: Z-Wave Plus version
- **roleType**: Z-Wave Plus role type
- **roleTypeString**: Z-Wave Plus role type description
- **nodeType**: Z-Wave Plus node type
- **installerIcon**: Icon for installer
- **userIcon**: Icon for user

Command ZWavePlusInfo Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_zwave_plus_info_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send ZWave+ Info Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.43 Command Class Firmware Update (0x7A/122)

Version 4, Controlled

Allows to update firmware of the device (OTA, Over-The-Air upgrade).

Data holders:

- **upgradeable**: Flag representing if the firmware is upgradable
- **firmwareCount**: Number of firmwares that can be updated using OTA (for multi chip devices, 0 is Z-Wave chip only)
- **updateStatus**: Indicated the status of the update process
- **waitTime**: Time the device will take before rebooting with newly upgraded firmware
- **manufacturerId**: Manufacturere ID
- **firmwareId**: Firmware Id
- **firmware[n]**: Firmware Id of firmware [n]
- **checksum**: Checksum of the firmware
- **fragmentTransmitted**: Number of fragments transmitted (useful to make progress bar)
- **fragmentCount**: Number of fragments to be transmitted in total (useful to make progress bar)
- **fragmentSize**: Internal
- **firmwareData**: Internal

Command FirmwareUpdate Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_firmware_update_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Firmware Metadata Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: upgradeable, firmwareCount, updateStatus, manufacturerId, manufacturerId, firmwareId, firmware[n], checksum updated

Command FirmwareUpdate Perform

Syntax: Perform(manufacturerId, firmwareId, firmwareTarget, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_firmware_update_perform(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, int manufacturerId, int firmwareId, ZWBYTE firmwareTarget, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Firmware Update Request Get. On process start Z-Way sets fragmentCount:. devices.N.instances.0.commandClasses.122.data.fragmentCount = 3073 (0x00000c01). Then it asks the device to start the process. The device can refuse it (i.e. if local confirmation timed out). If confirmed, the device will send us a report with adjusted fragment size (if it wants Z-Way to send by smaller packets) and report "Ready" (updateStatus = 255, see below). devices.N.instances.0.commandClasses.122.data.updateStatus = 255 (0x000000ff). devices.N.instances.0.commandClasses.122.data.fragmentCount = 3277 (0x00000ccd). At this point fragmentTransmitted == 0. devices.N.instances.0.commandClasses.122.data.fragmentTransmitted =0. Then device starts asking Z-Way for different packets. Z-Way will update fragmentTransmitted to allow track the process. Once done (fragmentCount == fragmentTransmitted), the device will send again a report if the flashing was successful. updateStatus contains the status: checksum error = 0, assemble error = 1, success, restart manually = 254, success, automatic restart = 255. waitTime refers to the time device will take to reboot. devices.N.instances.0.commandClasses.122.data.updateStatus = 255 (0x000000ff). devices.N.instances.0.commandClasses.122.data.waitTime = 5 (0x00000005)

Parameter manufacturerId: Manufacturer Id (2 bytes)

Parameter firmwareId: Firmware Id (2 bytes)

Parameter firmwareTarget: Firmware target number (0 for main chip, 1..255 for additional chips). Used only for CC v3 and above

Parameter data: Firmware image data in binary format (use hex2bin to convert from Intel Hex)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: updateStatus, waitTime, fragmentCount, fragmentTransmitted updated

B.44 Command Class Association Group Information (0x59/89)

Version 1, Supported and Controlled

Describes association groups defined by Association Command Class and command sent to group members.

Data holders:

- **[groupId]**: Subtree for grouId
- **groupName**: Group name
- **profile**: Group profile Id
- **mode**: Internal. Reserved.
- **eventCode**: Internal. Reserved
- **commands**: Subtree for commands
- **[commandClassId]**: Command Class Id of the command sent to group members
- **[commandId]**: Command Id corresponding to Command Class Id
- **dynamic**: Flag describing if the list can change and periodic request to update information is suggested

Command AssociationGroupInformation GetInfo

Syntax: GetInfo(groupId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_group_information_get_info(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE groupId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send AGI Get Info

Parameter groupId: Group Id to get info for (0 for all groups)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command AssociationGroupInformation GetName

Syntax: GetName(groupId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_group_information_get_name(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE groupId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send AGI Get Name

Parameter groupId: Group Id to get info for (0 for all groups)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command AssociationGroupInformation GetCommands

Syntax: GetCommands(groupId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_association_group_information_get_commands(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE groupId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send AGI Get Commands

Parameter groupId: Group Id to get info for (0 for all groups)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.45 Command Class SwitchColor (0x33/51)

Version 3, Controlled

Allows to control color for multicolor lights including LED bulbs and LED strips. Device reports it's capabilities:.
- 0 Warm White (0x00...0xFF: 0...100%).
- 1 Cold White (0x00:...0xFF: 0...100%).
- 2 Red (0x00...0xFF: 0...100%).
- 3 Green (0x00...0xFF: 0...100%).
- 4 Blue (0x00...0xFF: 0...100%).
- 5 Amber (for 6ch Color mixing) (0x00...0xFF: 0...100%).
- 6 Cyan (for 6ch Color mixing) (0x00...0xFF: 0...100%).
- 7 Purple (for 6ch Color mixing) (0x00...0xFF: 0...100%).
- 8 Indexed Color (0x00...0x0FF: Color Index 0...255).

Data holders:

- **capabilityMask**: Internal. Bit mask with supported capabilities
- **[capabilityId]**: Subtree for capabilityId
- **capabilityString**: Capability description
- **level**: Level of capability

Command SwitchColor Get

Syntax: Get(capabilityId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_color_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE capabilityId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchColor Get

Parameter capabilityId: Capability Id

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SwitchColor Set

Syntax: Set(capabilityId, state, duration = 0xff, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_color_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE capabilityId, ZWBYTE state, ZWBYTE duration, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchColor Set

Parameter capabilityId: Capability Id

Parameter state: State to be set for the capability

Parameter duration: Duration of change:. 0 instantly. 0x01...0x7f in seconds. 0x80...0xfe in minutes mapped to 1...127 (value 0x80=128 is 1 minute). 0xff use device factory default

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SwitchColor SetMultiple

Syntax: SetMultiple(capabilityIds, states, duration = 0xff, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_color_set_multiple(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, const ZWBYTE * capabilityIds, const ZWBYTE * states, ZWBYTE duration, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchColor SetMultiple

Parameter capabilityIds: Array of capabilities to set

Parameter states: Array of state values to be set for the capabilities

Parameter duration: Duration of change:. 0 instantly. 0x01...0x7f in seconds. 0x80...0xfe in minutes mapped to 1...127 (value 0x80=128 is 1 minute). 0xff use device factory default

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SwitchColor StartStateChange

Syntax: StartStateChange(capabilityId, dir, ignoreStartLevel = TRUE, startLevel = 50, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_color_start_state_change(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE capabilityId, ZWBYTE dir, ZWBOOL ignoreStartLevel, ZWBYTE startLevel, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchColor StartStateChange

Parameter capabilityId: Capability Id to start changing state for

Parameter dir: Direction of change: 0 to incrase, 1 to decrase

Parameter ignoreStartLevel: If set to True, device will ignore start level value and will use it's curent value

Parameter startLevel: Start level to change from

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SwitchColor StopStateChange

Syntax: StopStateChange(capabilityId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_switch_color_stop_state_change(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE capabilityId, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SwitchColor StopStateChange

Parameter capabilityId: Capability Id to stop changing state for

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.46 Command Class SoundSwitch (0x79/121)

Version 2, Controlled

Allows to play tones and configure volume.

Data holders:

- **currentTone**: Currently played tone Id
- **currentVolume**: The volume of the currently played tone
- **defaultTone**: Default tone Id

- **defaultVolume**: Default volume
- **tonesNumber**: Number of tones
- **[toneId]**: Subtree for toneId
- **toneName**: Tone name
- **duration**: Duration in seconds

Command SoundSwitch TonePlayGet

Syntax: TonePlayGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sound_switch_tone_play_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SoundSwitch TonePlayGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SoundSwitch TonePlaySet

Syntax: TonePlaySet(toneId, volume = 0x00, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sound_switch_tone_play_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE toneId, ZWBYTE volume, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SoundSwitch TonePlaySet

Parameter toneId: Tone Id. 1...254 tone Id. 255 play default tone

Parameter volume: Tone volume:. 0 use default. 1...100 specific volume. 255 last used volume

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SoundSwitch ConfigurationGet

Syntax: ConfigurationGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sound_switch_configuration_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SoundSwitch ConfigurationGet

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command SoundSwitch ConfigurationSet

Syntax: ConfigurationSet(toneId, volume = 0x00, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_sound_switch_configuration_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE toneId, ZWBYTE volume, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SoundSwitch ConfigurationSet

Parameter toneId: Tone Id. 1...254 tone Id. 255 play default tone

Parameter volume: Tone volume:. 0 use default. 1...100 specific volume. 255 last used volume

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.47 Command Class BarrierOperator (0x66/102)

Version 1, Controlled

Allows to control barriers and garage doors as well as their signal lamps.

Data holders:

- **state**: Barrier state
- **signalMask**: Internal. Bit mask of available signals
- **[signalId]**: Subtree for signal
- **signalTypeString**: Signal description
- **state**: Signal state

Command BarrierOperator Get

Syntax: Get(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_barrier_operator_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send BarrierOperator Get

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command BarrierOperator Set

Syntax: Set(state, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_barrier_operator_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE state, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send BarrierOperator Set

Parameter state: State to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command BarrierOperator SignalGet

Syntax: SignalGet(signalType, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_barrier_operator_signal_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE signalType, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send BarrierOperator Signal Get

Parameter signalType: Signal subsystem type to get state for

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command BarrierOperator SignalSet

Syntax: SignalSet(signalType, state, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_barrier_operator_signal_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE signalType, ZWBYTE state, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send BarrierOperator Signal Set

Parameter signalType: Signal subsystem type to set state for

Parameter state: State to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.48 Command Class SimpleAVControl (0x94/148)

Version 4, Supported and Controlled

Allows to control A/V devices.

Data holders:

- **bitmask**: Bit mask with supported keys. Refer to Expert UI pyzw_zwave.js or Silicon Labs documentation for description of buttons.
- **bitmasks**: Internal
- **sequenceNumber**: Internal
- **reportsNumber**: Internal

Command SimpleAVControl Set

Syntax: Set(keyAttribute, avCommand, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_simple_av_control_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE keyAttribute, unsigned int avCommand, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SimpleAVControl Set

Parameter keyAttribute: 0 for key Down, 1 for key Up, 2 for key Alive (repeated every 100...200 ms)

Parameter avCommand: Command to be sent. One of 465 predefined in Z-Wave protocol

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.49 Command Class Security (0x98/152)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library. There are no functions to execute.

Data holders:

- **controller->data->secureControllerId**: Node Id of secure controller: node that established secure channel when we are secondary controller (this data is on controller data tree)
- **device->data->secureChannelEstablished**: Flag describing if security interview was successful and secure channel is established (this data is on device data tree)
- **secureNodeInfoFrame**: Secure Node Information Frame
- **securityAbandoned**: Security interview failed
- **scheme**: Secure scheme supported
- **securityRequested**: Internal
- **rNonce**: Internal
- **rNonceAckWait**: Internal
- **canStream**: Internal
- **firstPart**: Internal
- **sequenceId**: Internal
- **toFollow**: Internal

Command Security Inject

Syntax: Inject(data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_security_inject(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Security Inject

Parameter data: Data to set

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.50 Command Class SecurityS2 (0x9F/159)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library. There are no functions to execute.

Data holders:

- **ctrlDevice->data->secureControllerId**: Node Id of secure controller: node that established secure channel when we are secondary controller (this data is on controller data tree)
- **device->data->secureChannelEstablished**: Flag describing if security interview was successful and secure channel is established (this data is on device data tree)
- **secureNodeInfoFrames**: Secure Node Information Frames for each security level
- **securityAbandoned**: Security interview failed
- **securityRequested**: Internal
- **requestedKeys**: Subtree of requested keys
- **S0**: True if requested
- **S2Unauthenticated**: True if requested
- **S2Authenticated**: True if requested
- **S2Access**: True if requested
- **grantedKeys**: Subtree of granted keys
- **S0**: True if granted
- **S2Unauthenticated**: True if granted
- **S2Authenticated**: True if granted
- **S2Access**: True if granted
- **device->data->securityS2ExchangedKeys**: Mask of granted keys (for re-interview)
- **publicKey**: Public key of the device
- **publicKeyVerified**: Verified public key of the device by the user
- **publicKeyAuthenticationRequired**: True if the user have to verify
- **controller->data->S2RequireCSA**: Require Client Side Authentication during joining network as secondary

Command SecurityS2 Inject

Syntax: Inject(data, keyClass, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_security_s2_inject(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, const ZWBYTE * data, ZWBYTE keyClass, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send Security S2 Inject

Parameter data: Data to set

Parameter keyClass: Security S2 key class

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.51 Command Class EntryControl (0x6F/111)

Version 1, Controlled

Receive codes from entry control devices like security panels.

Data holders:

- **sequence**: Sequence number to prevent duplicates. Internal
- **event**: Event Id
- **eventData**: Additional data to the event
- **keys**: Array with supported keys Id
- **dataTypes**: Array with supported data types
- **eventTypes**: Array with supported event type Id
- **keyCacheSize**: Key cache size before sending to the controller
- **keyCacheSizeMin**: Minumum supported keyCacheSize
- **keyCacheSizeMax**: Maxumum supported keyCacheSize
- **keyCacheTimeout**: Key cache timeout before sending to the controller, in seconds
- **keyCacheTimeoutMin**: Minumum supported keyCacheTimeout
- **keyCacheTimeoutMax**: Maxumum supported keyCacheTimeout

Command EntryControl ConfigurationGet

Syntax: ConfigurationGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_entry_control_configuration_get(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request keyCacheSize and keyCacheTimeout

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Command EntryControl ConfigurationSet

Syntax: ConfigurationSet(keyCacheSize, keyCacheTimeout, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_cc_entry_control_configuration_set(ZWay zway, ZWBYTE node_id, ZWBYTE instance_id, ZWBYTE keyCacheSize, ZWBYTE keyCacheTimeout, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set keyCacheSize and keyCacheTimeout

Parameter keyCacheSize: Key cache size before sending to the controller

Parameter keyCacheTimeout: Key cache timeout before sending to the controller, in seconds

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

B.52 Command Class CRC16 (0x56/86)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library to use better 16 bits packet checksum. There are no functions to execute.

Data holders:

- **crc16Requested**: Internal

B.53 Command Class MultiCmd (0x8F/143)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library to save battery life time. There are no functions to execute.

Data holders:

- **maxNum**: Max number of packets to be encapsulated. Can be tunned to lower (to workaround buggy devices, 1 to turn off) or rise (to get bettery performance)

B.54 Command Class Supervision (0x6C/108)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library to guarantee delivery report on every command (even on Set). There are no functions to execute.

Data holders:

- **[sessionId]**: Subtree with session status
- **status**: Current session status (0 = Not supported, 1 = Working, 2 = Fail, 3 = Busy, 255 = Success)
- **duration**: Expected time to finish the operation
- **moreStatusUpdates**: True if more updates on the session status are expected
- **lastSession**: Internal

B.55 Command Class Application Status (0x22/34)

Version 1, Supported and Controlled

This Command Class is transparently implemented in the library to retry on device Busy report. There are no functions to execute.

B.56 Command Class Version (0x86/134)

Version 2, Supported and Controlled

Allows to get version of each Command Class supported by the device as well as firmware version.

Data holders:

- **commandClass->data->version**: Version of specific Command Class (this data is on Command Class data tree)
- **ZWLib**: SDK library type
- **ZWProtocolMajor**: SDK version major
- **ZWProtocolMinor**: SDK version minor
- **SDK**: SDK description
- **applicationMajor**: Application version major
- **applicationMinor**: Application version minor
- **hardwareVersion**: Hardware revision of the device
- **firmwareCount**: Number of chips (firmwares) in the device (excluding Z-Wave chip)
- **[firmwareId]**: Subtree for firmwareId information
- **major**: Additional chip application version major
- **minor**: Additional chip application version major

B.57 Command Class DeviceResetLocally (0x5A/90)

Version 1, Supported and Controlled

Reports to the controller that device was resetted locally (using local button operation).

Data holders:

- **reset**: Becomes True if the device sent us DeviceResetLocally notification. This means the device is certainly not in our network anymore

B.58 Command Class Central Scene (0x5B/91)

Version 3, Supported and Controlled

Allows to receive central controller oriented scene actions. Scenes are triggered by pushing a button on a remote control or wall controller. Note that Z-Way supports only V1, but in most cases you don't need it to be enabled in the NIF. Controlled version is V3.

Data holders:

- **maxScenes**: Number of scenes supported
- **slowRefreshSupport**: Flag to indicate if the device supports Slow Refresh mode
- **slowRefresh**: Flag to indicate if the device is currently in Slow Refresh mode
- **currentScene**: Last activated scene
- **keyAttribute**: Button (or key) action: 0 for key press, 1 for key release, 2 for key held down (should bre repeated at least every 200ms)
- **sequence**: Internal. To ignore duplicate packats.
- **sceneSupportedKeyAttributesMask**: Holds the list of supported key attributes for each scene
- **[sceneId]**: Array of supported key attributes for Scene Id: 0 for 1 press, 1 for release after hold, 2 for hold, 3..6 for 2..5 presses

C Function Class Reference

Function Class GetSerialAPICapabilities

Syntax: GetSerialAPICapabilities(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_serial_api_capabilities(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request Serial API capabilities

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: capabilities, manufacturerId, manufacturerProductId, manufacturerProductType, APIVersion, vendor

Function Class SerialAPISetTimeouts

Syntax: SerialAPISetTimeouts(ackTimeout, byteTimeout, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_set_timeouts(ZWay zway, ZWBYTE ackTimeout, ZWBYTE byteTimeout, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set Serial API timeouts

Parameter ackTimeout: Time for the stick to wait for ACK (in 10ms units)

Parameter byteTimeout: Time for the stick to assemble a full packet (in 10ms units)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: curSerialAPIAckTimeout10ms, curSerialAPIByteTimeout10ms, oldSerialAPIAckTimeout10ms, oldSerialAPIByteTimeout10ms

Function Class SerialAPIGetInitData

Syntax: SerialAPIGetInitData(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_get_init_data(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request initial information about devices in network

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: ZWVersion, ZWaveChip, list of Z-Wave devices is generated

Function Class SerialAPIApplicationNodeInfo

Syntax: SerialAPIApplicationNodeInfo(listening, optional, flirs1000, flirs250, genericClass, specificClass, nif, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_application_node_info(ZWay zway, ZWBOOL listening, ZWBOOL optional, ZWBOOL flirs1000, ZWBOOL flirs250, ZWBYTE generic_class, ZWBYTE specific_class, const ZWBYTE * nif, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set controller node information

Parameter listening: Listening flag

Parameter optional: Optional flag (set if device supports more CCs than described as mandatory for it's Device Type)

Parameter flirs1000: FLiRS 1000 flag (hardware have to be based on FLiRS library to support it)

Parameter flirs250: FLiRS 250 flag (hardware have to be based on FLiRS library to support it)

Parameter genericClass: Generic Device Type

Parameter specificClass: Specific Device Type

Parameter nif: New NIF

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class WatchDogStart

Syntax: WatchDogStart(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_watchdog_start(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Start WatchDog

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class WatchDogStop

Syntax: WatchDogStop(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_watchdog_stop(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Stop WatchDog

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class GetHomeId

Syntax: GetHomeId(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_home_id(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request Home Id and controller Node Id

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: homeId, nodeId

Function Class GetControllerCapabilities

Syntax: GetControllerCapabilities(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_controller_capabilities(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request controller capabilities (primary role, SUC/SIS availability)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: isInOthersNetwork, isPrimary, isRealPrimary, isSUC, isSUC, SISPresent

Function Class GetVersion

Syntax: GetVersion(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_version(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request controller hardware version

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: ZWLib, ZWProtocolMajor, ZWProtocolMinor, libType, SDK, devices[ctrlId].data.ZWLib, devices[ctrlId].data.ZWProto devices[ctrlId].data.ZWProtocolMinor, devices[ctrlId].data.SDK

Function Class GetSUCNodeId

Syntax: GetSUCNodeId(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_suc_node_id(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request SUC Node Id

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: SUCNodeId

Function Class EnableSUC

Syntax: EnableSUC(enable, sis, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_enable_suc(ZWay zway, ZWBOOL enable, ZWBOOL sis, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Enable or disable SUC/SIS functionality of the controller

Parameter enable: True to enable functionality, False to disable

Parameter sis: True to enable SIS functionality, False to enable SUC only

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SetSUCNodeId

Syntax: SetSUCNodeId(nodeId, enable, sis, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_set_suc_node_id(ZWay zway, ZWBYTE node_id, ZWBOOL enable, ZWBOOL sis, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Assign new SUC/SIS or disable existing

Parameter nodeId: Node Id to be assigned/disabled as SUC/SIS

Parameter enable: True to enable, False to disable

Parameter sis: True to assign SIS role, False to enable SUC role only

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class MemoryGetByte

Syntax: MemoryGetByte(offset, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_memory_get_byte(ZWay zway, unsigned short offset, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Read single byte from EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: memoryGetData, memoryGetAddress

Function Class MemoryGetBuffer

Syntax: MemoryGetBuffer(offset, length, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_memory_get_buffer(ZWay zway, unsigned short offset, ZWBYTE length, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Read multiple bytes from EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter length: Number of byte to be read

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: memoryGetData, memoryGetAddress

Function Class MemoryPutByte

Syntax: MemoryPutByte(offset, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_memory_put_byte(ZWay zway, unsigned short offset, ZWBYTE data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Write single byte to EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter data: Byte to be written

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class MemoryPutBuffer

Syntax: MemoryPutBuffer(offset, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_memory_put_buffer(ZWay zway, unsigned short offset, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Write multiple bytes to EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter data: Bytes to be written

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class NVMGetId

Syntax: NVMGetId(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_nvm_get_id(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Read type of extended EEPROM

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: memoryCapacity, memoryManufacturerId, memoryType

Function Class NVMExtReadLongByte

Syntax: NVMExtReadLongByte(offset, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_nvm_ext_read_long_byte(ZWay zway, unsigned int offset, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Read single byte from extended EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: memoryGetData, memoryGetAddress

Function Class NVMExtReadLongBuffer

Syntax: NVMExtReadLongBuffer(offset, length, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_nvm_ext_read_long_buffer(ZWay zway, unsigned int offset, unsigned short length, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Read multiple bytes from exended EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter length: Number of byte to be read

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: memoryGetData, memoryGetAddress

Function Class NVMExtWriteLongByte

Syntax: NVMExtWriteLongByte(offset, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_nvm_ext_write_long_byte(ZWay zway, unsigned int offset, ZWBYTE data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Write single byte to extended EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter data: Byte to be written

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class NVMExtWriteLongBuffer

Syntax: NVMExtWriteLongBuffer(offset, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_nvm_ext_write_long_buffer(ZWay zway, unsigned int offset, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Write multiple bytes to extended EEPROM

Parameter offset: Offset in application memory in EEPROM

Parameter data: Bytes to be written

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class IsFailedNode

Syntax: IsFailedNode(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_is_failed_node(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Checks if node is failed

Parameter nodeId: Node Id to be checked

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: device[node_id].data.isFailed

Function Class SendDataAbort

Syntax: SendDataAbort(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_data_abort(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Abort send data. Note that this function works unpredictably in multi callback environment !

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SerialAPISoftReset

Syntax: SerialAPISoftReset(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_soft_reset(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Soft reset. Restarts Z-Wave chip

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SendData

Syntax: SendData(nodeId, data, description = NULL, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_data(ZWay zway, ZWBYTE node_id, const ZWBYTE * data, ZWCSTR description, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send data. Packets are sent in AUTO_ROUTE mode with EXPLRER_FRAME enabled for listening devices (ignored if not supported by the hardware [based on 5.0x branch])

Parameter nodeId: Destination Node Id (NODE_BROADCAST to send non-routed broadcast packet)

Parameter data: Paket payload

Parameter description: Packet description for queue inspector and logging

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.lastSend

Function Class SendDataSecure

Syntax: SendDataSecure(nodeId, data, description = NULL, keyClass = 0, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_data_secure(ZWay zway, ZWBYTE node_id, const ZWBYTE * data, ZWCSTR description, ZWBYTE key_class, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send data using security. Packets are sent in AUTO_ROUTE mode with EXPLRER_FRAME enabled for listening devices (ignored if not supported by the hardware [based on 5.0x branch]). Explicitelly use security

Parameter nodeId: Destination Node Id (NODE_BROADCAST to send non-routed broadcast packet)

Parameter data: Paket payload

Parameter description: Packet description for queue inspector and logging

Parameter keyClass: Security class to use: 0 - S0, 1 - S2 Unauthenticated, 2 - S2 Authenticated, 4 - S2 Access

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.lastSend

Function Class GetNodeProtocolInfo

Syntax: GetNodeProtocolInfo(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_node_protocol_info(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get node protocol info

Parameter nodeId: Node Id of the device in question

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.isListening, devices[node_id].data.isRouting, devices[node_id].data.optional, devices[node_id].data.sensor1000, devices[node_id].data.sensor250, devices[node_id].data.infoProtocolSpecific

Function Class GetRoutingTableLine

Syntax: GetRoutingTableLine(nodeId, removeBad = FALSE, removeRepeaters = FALSE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_routing_table_line(ZWay zway, ZWBYTE node_id, ZWBOOL remove_bad, ZWBOOL remove_repeaters, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get routing table line

Parameter nodeId: Node Id of the device in question

Parameter removeBad: Exclude failed nodes from the listing

Parameter removeRepeaters: Exclude repeater nodes from the listing

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.neighbours

Function Class AssignReturnRoute

Syntax: AssignReturnRoute(nodeId, destId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_assign_return_route(ZWay zway, ZWBYTE node_id, ZWBYTE dest_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Assign return route to specified node

Parameter nodeId: Node Id of the device that have to store new route

Parameter destId: Destination Node Id of the route

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class AssignSUCReturnRoute

Syntax: AssignSUCReturnRoute(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_assign_suc_return_route(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Assign return route to SUC

Parameter nodeId: Node Id of the device that have to store route to SUC

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class DeleteReturnRoute

Syntax: DeleteReturnRoute(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_delete_return_route(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Delete return route

Parameter nodeId: Node Id of the device that have to delete all assigned return routes

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class DeleteSUCReturnRoute

Syntax: DeleteSUCReturnRoute(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_delete_suc_return_route(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Delete return route to SUC

Parameter nodeId: Node Id of the device that have to delete route to SUC

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SetDefault

Syntax: SetDefault(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_set_default(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Reset the controller. Note: this function will delete ALL data from the Z-Wave chip and restore it to factory default !. Sticks based on 4.5x and 6.x SDKs will also generate a new Home Id

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: lastExcludedDevice, lastIncludedDevice

Function Class SendSUCNodeId

Syntax: SendSUCNodeId(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_suc_node_id(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send SUC Node Id. Informs portable and static controllers about new or deleted SUC/SIS

Parameter nodeId: Node Id of the device that have to be informed about new or deleted SIC/SIS

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SendNodeInformation

Syntax: SendNodeInformation(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_node_information(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send NIF of the stick

Parameter nodeId: Destination Node Id (NODE_BROADCAST to send non-routed broadcast packet)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class RequestNodeInformation

Syntax: RequestNodeInformation(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_request_node_information(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request NIF of a device

Parameter nodeId: Node Id to be requested for a NIF

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.nodeInfoFrame

Function Class RemoveFailedNode

Syntax: RemoveFailedNode(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_remove_failed_node(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Remove failed node from network. Before removing SDK will check that the device is really unreachable

Parameter nodeId: Node Id to be removed from network

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: lastExcludedDevice

Function Class ReplaceFailedNode

Syntax: ReplaceFailedNode(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_replace_failed_node(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Replace failed node with a new one. Be ware that a failed node can be replaced by a node of another type. This can lead to probles!. Always request device NIF and force re-interview after successful replace process

Parameter nodeId: Node Id to be replaced by new one

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: lastIncludedDevice, controllerState

Function Class RequestNetworkUpdate

Syntax: RequestNetworkUpdate(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_request_network_update(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request network topology update from SUC/SIS. Note that this process may also fail due more than 64 changes from last sync. In this case a re-inclusion of the controller (self) is required

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class RequestNodeNeighbourUpdate

Syntax: RequestNodeNeighbourUpdate(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_request_node_neighbour_update(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request neighbours update for specific node

Parameter nodeId: Node Id to be requested for it's neighbours

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: devices[node_id].data.neighbours

Function Class SetLearnMode

Syntax: SetLearnMode(startStop, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_set_learn_mode(ZWay zway, ZWBOOL startStop, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set/stop Learn mode. Tries first classical inclusion then falls back to NWI automatically. Use zway_controller_set_learn_mode instead to get correctly set up the environment after inclusion

Parameter startStop: Start Learn mode if True, stop if False

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class AddNodeToNetwork

Syntax: AddNodeToNetwork(startStop, highPower = TRUE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_add_node_to_network(ZWay zway, ZWBOOL startStop, ZWBOOL highPower, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Start/stop Inclusion of a new node. Available on primary and inclusion controllers

Parameter startStop: Start inclusion mode if True, stop if False

Parameter highPower: Use full power during this operation if True. On False use low power mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class SmartStartEnable

Syntax: SmartStartEnable(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_smart_start_enable(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Enable Smart Start mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class RemoveNodeFromNetwork

Syntax: RemoveNodeFromNetwork(startStop, highPower = FALSE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_remove_node_from_network(ZWay zway, ZWBOOL startStop, ZWBOOL highPower, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Start/stop exclusion of a node. Note that this function can be used to exclude a device from previous network before including in ours. Available on primary and inclusion controllers

Parameter startStop: Start exclusion mode if True, stop if False

Parameter highPower: Use full power during this operation if True. On False use low power mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class RemoveNodeIdFromNetwork

Syntax: RemoveNodeIdFromNetwork(nodeId, startStop, highPower = FALSE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_remove_node_id_from_network(ZWay zway, ZWBYTE nodeId, ZWBOOL startStop, ZWBOOL highPower, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Start/stop exclusion of a node id. Note that this function can be used to exclude a device from previous network before including in ours. Available on primary and inclusion controllers

Parameter nodeId: NodeId to exclude. If 0 or > 232, any node will be excluded (like with zway_fc_remove_node_from_network / RemoveNodeFromNetwork)

Parameter startStop: Start exclusion mode if True, stop if False

Parameter highPower: Use full power during this operation if True. On False use low power mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class ControllerChange

Syntax: ControllerChange(startStop, highPower = TRUE, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_controller_change(ZWay zway, ZWBOOL startStop, ZWBOOL highPower, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set new primary controller (also known as Controller Shift). Same as Inclusion, but the newly included device will get the role of primary. Available only on primary controller

Parameter startStop: Start controller shift mode if True, stop if False

Parameter highPower: Use full power during this operation if True. On False use low power mode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class CreateNewPrimary

Syntax: CreateNewPrimary(startStop, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_create_new_primary(ZWay zway, ZWBOOL startStop, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Create new primary controller by SUC controller. Same as Inclusion, but the newly included device will get the role of primary. Available only on SUC. Be careful not to create two primary controllers! This can lead to network malfunction!

Parameter startStop: Start create new primary mode if True, stop if False

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: controllerState, lastExcludedDevice, lastIncludedDevice

Function Class ZMEFreqChange

Syntax: ZMEFreqChange(freq, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_freq_change(ZWay zway, ZWBYTE freq, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Change Z-Wave.Me Z-Stick 4 frequency. This function is specific for Z-Wave.Me hardware

Parameter freq: 0x01 RU. 0x02 IN. 0x03 US. 0x04 ANZ. 0x05 HK. 0x06 CN. 0x07 JP. 0x08 KR. 0x09 IL. 0x0A MY. 0xFF request current frequency (ZME firmwares prior to 5.03 don't support this feature)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: frequency

Function Class RFPowerLevelSet

Syntax: RFPowerLevelSet(level, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_rf_power_level_set(ZWay zway, ZWBYTE level, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set RF power level to specified value

Parameter level: 0 to 9

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class RFPowerLevelGet

Syntax: RFPowerLevelGet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_rf_power_level_get(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get RF power level current value

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SendTestFrame

Syntax: SendTestFrame(nodeId, level, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_send_test_frame(ZWay zway, ZWBYTE node_id, ZWBYTE level, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Send test frame to a node at a specified RF level

Parameter nodeId: Node Id to make test against

Parameter level: 0 to 9

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class FlashAutoProgSet

Syntax: FlashAutoProgSet(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_flash_auto_prog_set(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Put Z-Wave chip in Atuo Prog mode for USB/UART reflashing

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ExploreRequestInclusion

Syntax: ExploreRequestInclusion(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_explore_request_inclusion(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request NWI. Called from SetLearnMode

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ExploreRequestExclusion

Syntax: ExploreRequestExclusion(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_explore_request_exclusion(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Request NWE

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMEBootloaderFlash

Syntax: ZMEBootloaderFlash(seg, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_bootloader_flash(ZWay zway, ZWBYTE seg, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Start reflashing bootloader of Z-Wave.Me firmware for 5th generation Z-Wave chip. This function is specific for Z-Wave.Me hardware

Parameter seg: address of new bootloader location in 2K segments

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMEBootloaderLoadFlash

Syntax: ZMEBootloaderLoadFlash(data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_bootloader_load_flash(ZWay zway, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Load and start reflashing bootloader of Z-Wave.Me firmware for 7th generation Z-Wave chip. This function is specific for Z-Wave.Me hardware

Parameter data: Firmware image data in binary format (use hex2bin to convert from Intel Hex)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMECapabilities

Syntax: ZMECapabilities(data = NULL, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_capabilities(ZWay zway, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get or set firmware capabilities. This function is specific for Z-Wave.Me hardware

Parameter data: data to set (NULL to get)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: caps, uuid

Function Class ZMELicenseSet

Syntax: ZMELicenseSet(data = NULL, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_license_set(ZWay zway, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set Z-Wave.Me firmware capabilities. This function is specific for Z-Wave.Me hardware

Parameter data: data to set (NULL to get)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: caps

Function Class ZMEPHISetLED

Syntax: ZMEPHISetLED(led, status, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephisetled(ZWay zway, ZWBYTE led, ZWBYTE status, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Activate LEDs on Philio hw. This function is specific for Philio hardware

Parameter led: LED id: 0x10 (Logo), 0x11 (Around), 0x12 (Misc)

Parameter status: LED status 2 (Off), 4 (On), 8 (Flash), 16 (Slow flash), 32 (Slow dimming)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMEPHIGetButton

Syntax: ZMEPHIGetButton(button, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephigetbutton(ZWay zway, ZWBYTE button, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get button state on Philio hw. This function is specific for Philio hardware

Parameter button: 0: Tamper Key, 1: Function Key A, 2: Function Key B

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: philiohw.tamper, philiohw.funcA, philiohw.funcB

Function Class ZMEPHIGetPower

Syntax: ZMEPHIGetPower(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephigetpower(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get power state on Philio hw. This function is specific for Philio hardware

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: philiohw.powerFail, philiohw.batteryLevel, philiohw.charging, philiohw.batteryFail

Function Class ZMEPHIGetBattery

Syntax: ZMEPHIGetBattery(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephigetbattery(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get battery state on Philio hw. This function is specific for Philio hardware

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: philiohw.batteryADCLevel

Function Class ZMEPHIGetRTC

Syntax: ZMEPHIGetRTC(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephigetrtc(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get RTC from Philio hw. This function is specific for Philio hardware

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Report: set system time

Function Class ZMEPHISetRTC

Syntax: ZMEPHISetRTC(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zmephisetrtc(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set RTC on Philio hw. This function is specific for Philio hardware

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class InjectPacket

Syntax: InjectPacket(nodeId, data, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_application_command_handler_inject(ZWay zway, ZWBYTE node_id, const ZWBYTE * data, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Inject command in Z-Way as it was received via Z-Wave. This function is for debugging only

Parameter nodeId: Source Node Id

Parameter data: Paket payload (should looks like ccId, ccCmd, data, ...)

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class GetBackgroundRSSI

Syntax: GetBackgroundRSSI(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_background_rssi(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get background noise level

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SerialAPISetupSetIMA

Syntax: SerialAPISetupSetIMA(enable, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_setup_set_ima(ZWay zway, ZWBOOL enable, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Enable Advanced IMA in Z-Wave.Me firmware

Parameter enable: Set feature state: True to enable, False to disable

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SerialAPISetupSetTxPower

Syntax: SerialAPISetupSetTxPower(power, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_serial_api_setup_set_tx_power(ZWay zway, ZWBYTE power, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set Tx power in Z-Wave.Me firmware

Parameter power: Power level in 10*dBm

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ClearNetworkStats

Syntax: ClearNetworkStats(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_clear_network_stats(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Clear statistics gathered by the Z-Wave protocol

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class GetNetworkStats

Syntax: GetNetworkStats(successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_network_stats(ZWay zway, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get statistics gathered by the Z-Wave protocol

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class GetPriorityRoute

Syntax: GetPriorityRoute(nodeId, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_get_priority_route(ZWay zway, ZWBYTE node_id, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Get the route with the highest priority

Parameter nodeId: Node ID we are interested in

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SetPriorityRoute

Syntax: SetPriorityRoute(nodeId, repeater1, repeater2, repeater3, repeater4, routeSpeed, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_set_priority_route(ZWay zway, ZWBYTE node_id, ZWBYTE repeater1, ZWBYTE repeater2, ZWBYTE repeater3, ZWBYTE repeater4, ZWBYTE route_speed, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set the route with the highest priority

Parameter nodeId: Node ID we are interested in

Parameter repeater1: Hop #1 in the route. Value 0 means direct range. Values > 232 clears the priority route and LWR (Last Working Route is selected by the protocol)

Parameter repeater2: Hop #2 in the route. Value 0 means end of route

Parameter repeater3: Hop #3 in the route. Value 0 means end of route

Parameter repeater4: Hop #4 in the route. Value 0 means end of route

Parameter routeSpeed: Baudrate to use: 1 for 9.6 kbps, 2 for 40 kbps, 3 for 100 kbps

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class SetPromiscuousMode

Syntax: SetPromiscuousMode(enable, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_set_promiscuous_mode(ZWay zway, ZWBOOL enable, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Enable or disable promiscuous mode

Parameter enable: True to enable functionality, False to disable

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class AssignPriorityReturnRoute

Syntax: AssignPriorityReturnRoute(nodeId, destId, repeater1, repeater2, repeater3, repeater4, routeSpeed, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_assign_priority_return_route(ZWay zway, ZWBYTE node_id, ZWBYTE dest_id, ZWBYTE repeater1, ZWBYTE repeater2, ZWBYTE repeater3, ZWBYTE repeater4, ZWBYTE route_speed, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Assign priority return route to specified node

Parameter nodeId: Node Id of the device that have to store new route

Parameter destId: Destination Node Id of the route

Parameter repeater1: Hop #1 in the route. Value 0 means direct range

Parameter repeater2: Hop #2 in the route. Value 0 means end of route

Parameter repeater3: Hop #3 in the route. Value 0 means end of route

Parameter repeater4: Hop #4 in the route. Value 0 means end of route

Parameter routeSpeed: Baudrate to use: 1 for 9.6 kbps, 2 for 40 kbps, 3 for 100 kbps

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class AssignPrioritySUCReturnRoute

Syntax: AssignPrioritySUCReturnRoute(nodeId, repeater1, repeater2, repeater3, repeater4, routeSpeed, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_assign_priority_suc_return_route(ZWay zway, ZWBYTE node_id, ZWBYTE repeater1, ZWBYTE repeater2, ZWBYTE repeater3, ZWBYTE repeater4, ZWBYTE route_speed, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Assign priority return route to SUC

Parameter nodeId: Node Id of the device that have to store route to SUC

Parameter repeater1: Hop #1 in the route. Value 0 means direct range

Parameter repeater2: Hop #2 in the route. Value 0 means end of route

Parameter repeater3: Hop #3 in the route. Value 0 means end of route

Parameter repeater4: Hop #4 in the route. Value 0 means end of route

Parameter routeSpeed: Baudrate to use: 1 for 9.6 kbps, 2 for 40 kbps, 3 for 100 kbps

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMESerialAPIOptionsJammingDetectionSet

Syntax: ZMESerialAPIOptionsJammingDetectionSet(ch1Threshold, ch2Threshold, ch3Threshold, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_serialapi_options_jamming_detection_set(ZWay zway, ZWBYTE ch1_threshold, ZWBYTE ch2_threshold, ZWBYTE ch3_threshold, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set Jamming Detection threshold. Jamming detection fine tuning in NVM:. address 6530, 3 bytes: ch1_threshold ch2_threshold ch3_threshold. address 6533, 2 bytes: how many times should measurements exceed in last 10 seconds to trigger the alarm, default 2 times. address 6535, 2 bytes: repeat the alarm in N*0.1 seconds if the jamming persists, default 0100 = 25.6 sec

Parameter ch1Threshold: Threshold for channel 1

Parameter ch2Threshold: Threshold for channel 2

Parameter ch3Threshold: Threshold for channel 3

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMESerialAPIOptionsAdvancedIMASet

Syntax: ZMESerialAPIOptionsAdvancedIMASet(enable, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_serialapi_options_advanced_ima_set(ZWay zway, ZWBOOL enable, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Enable/disable advanced IMA by Z-Wave.Me that includes incoming route path information

Parameter enable: Turn on or off

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMESerialAPIOptionsStaticAPISet

Syntax: ZMESerialAPIOptionsStaticAPISet(enable, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_serialapi_options_static_api_set(ZWay zway, ZWBOOL enable, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Switch between Static and Bridge API for Z-Wave.Me firmware

Parameter enable: Turn on or off Static API

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

Function Class ZMESerialAPIOptionsUARTSpeedSet

Syntax: ZMESerialAPIOptionsUARTSpeedSet(speed = 115200, successCallback = NULL, failureCallback = NULL)

C Syntax: ZWError zway_fc_zme_serialapi_options_uart_speed_set(ZWay zway, speed_t speed, ZJobCustomCallback successCallback, ZJobCustomCallback failureCallback, void* callbackArg)

Description: Set high speed for UART interface for Z-Wave.Me firmware

Parameter speed: UART speed. Valid values are listed in UART_SPEEDS macro definition

Parameter successCallback: Custom function to be called on function success. NULL if callback is not needed

Parameter failureCallback: Custom function to be called on function failure. NULL if callback is not needed

D List of supported EnOcean devices

D.1 NodOn

- Wall Switch CWS-2-1-xx
- Soft Remote CRC-2-6-xx
- Card Switch CCS-2-1-01
- Door Window Sensor SDO-2-1-xx
- Temperature Sensor STP-2-1-xx

D.2 Thermokon

- Door Sensor SRW01
- Motion Detector SR-MOC
- Motion Detector SW-MOW
- Card Reader SR-KCS
- Wall Controller 55x55

D.3 Hubbel

- Motion Sensor wiSTAR OS
- Card Reader wiSATR Key Card
- Single Rocker Wall Controller wiStar
- Dual Rocker Wall Controller wiStar

D.4 AWAG

- Door Window Sensor FK101
- Card Reader KSM-CH
- Wall Controller WS-CH-102

D.5 Hoppe

- Door Sensor STM250

D.6 Schneider Elektrik

- Temperature Sensor SED-WDS
- Motion Detector SED-CMS
- Card Reader SED-KCS
- Wall Controller AED-1RS

D.7 PEHA

- Door Sensor FU FK
- Motion Detector FUB-BM
- Motion Detector FUB-BM DE
- Card Reader FU-BLS
- Wall Controller FU BLSN
- Wall Controller FU BLSJR
- Wall Controller FU BLSN-2
- Wall Controller FU BLSN

D.8 Eltako

- Wall Controller FT4-rw
- Motion detector FABH65
- Door Sensor FTKB-rw

D.9 EnOcean GmbH

- Door Sensor STM250
- Motion Detector EOSC
- Motion Detector EOSW
- Card Reader EKCS
- Single Rocker Wall Controller PTM210
- Dual Rocker Wall Controller PTM215

List of Tables

List of Figures