1-Wire Automation Server v1.1.0

User Manual – Part 1

June 2015

Table of Contents

Revision History

Date	Authors	Description
2015-03-06	Peter S'heeren	Initial release.
2015-06-16	Peter S'heeren	Added USBMicro U4x1 adapters. Added DS9097 adapter. Added W1. Added TMEX. Second release.

1 Overview

The 1-Wire Automation Server, called the server throughout this document, is a powerful software tool for automating 1-Wire-based projects.

The server has built-in drivers for many 1-Wire adapters that are available in today's market. A component called Device Nodes enables the server to control 1-Wire adapters and other interfaces like serial ports and I2C, in a generic and extensible way.

The server automates the acquisition of sensor data of 1-Wire slaves. The acquired data can be streamed to a database, summarized in graphical representations, etc.

The 1-Wire Automation Server is a true network server that allows clients to connect from remote systems. The server implements a client protocol that's easy to use both interactively (using a terminal program like PuTTY or netcat) as well as programmatically (from software).

The 1-Wire Automation Software includes a logger and a graphical front-end (GUI) application. Both are client programs that can connect with the server locally and remotely over the network. For example, you can run the server on a single board computer strategically places in the field, log sensor data on a database server system, and use the GUI program to monitor the server's state on a desktop PC.

2 Third-party Resources

LibUSB

The server depends on the LibUSB library for driving USB-based 1-Wire adapters. Currently these adapters include device on the DS2490 chip (like the DS9490B and DS9490R adapters), the Flyfish FF32 and the Flyfish FF34.

LibUSB is available for Linux and Windows. The server relies on the library in both systems.

In Linux, LibUSB works out of the box. You have to make sure that LibUSB is installed on your system. If LibUSB isn't installed, the server won't be able to work with USB-based 1-Wire adapters.

In Windows, you need to update the driver of the 1-Wire adapter to a specific driver in order to enable access through LibUSB. You can use the Zadig tool for updating the driver. You don't have to install the LibUSB library itself; the 1-Wire Automation Software includes the library.

LibUSB home page: http://www.libusb.org/

Zadig

This Windows tool allows you to update the driver of any USB device to a driver suitable for use with LibUSB. The tool includes various driver sets. It's recommended to install libusbK.

Home page: http://zadig.akeo.ie/

TMEX

TMEX offers an API for controlling 1-Wire adapters on Windows operating systems. TMEX is developed by Maxim Integrated.

Currently, TMEX supports maxim's 1-Wire adapters with USB, serial, passive serial, and parallel interfaces. The 1-Wire server supports all of these adapters except for the parallel to 1-Wire adapter.

SDK download page:

http://www.maximintegrated.com/en/products/ibutton/software/windowsdk/index.cfm

Drivers download page:

http://www.maximintegrated.com/en/products/ibutton/software/tmex/download_drivers.cfm

Note that TMEX comes with WinUSB version 7 while Zadig includes WinUSB version 6. If you have updated the drivers of your DS9490 adapter using Zadig, TMEX won't work for the adapter as it expect version 7. If this is the case, update to version 7 of WinUSB in the Device Manager.

3 Command Line

Parameter	Description
-service	Run the program as a service. For this to work, the program must be installed as a Windows service.
-console	Open a console. This parameter as no effect when the program is run as a service.
-v	Enable verbose output.
-h	Display help and exit.
-port n	The value specifies the port number the server must listen to. Value n=1..65535 decimal.
-romcode f	Set the default formatting style of ROM code in client responses. Valid values for f are **native** and **owfs**.
-cfg FILE	Specify configuration file.

Parameters **-service** and **-console** are specific to the Windows version of the server.

The **-port** parameter overrules the port number specified in the configuration file.

The **-romcode** parameter overrules the formatting style of ROM codes specified in the configuration file.

The **-cfg** parameter overrules the default configuration file. The given filename may include an absolute or relative path.

4 Running the Server

Current Directory

When the server starts up, it changes its current directory to the location where the server's executable file resides. This is also the directory where the 1-Wire Automation Software installer puts the program files, unless of course you've moved the files to another directory.

The current directory acts as the base directory for a filename that's specified with a relative path. This is true in the following places:

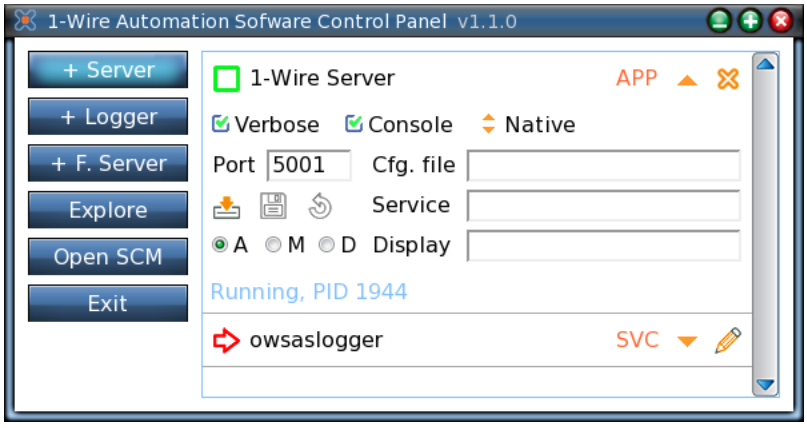
- Command line parameter **-cfg**.
- Configuration file keyword **cmdfile**.
- Client command **Topology Load**.
- Client command **Topology Save**.

Windows

Control Panel

The control panel application offers a convenient way of configuring and running the server (and logger). The 1-Wire Automation Software installer creates a shortcut on your desktop. When the control panel is running, it shows an icon in the desktop's tray for quick access.

The graphical interface enables you to run the server as an application or a Windows service. You can configure command line parameters, start and stop the server as application or Windows service, install the server as a Windows service, etc. Please read the 1-Wire Automation Software Control Panel user manual for more information.



Command Line

You can run the server from the command line. In this environment, you run the server as an application.

You can't run the server as a service from the command line; specifying parameter **-service** won't work. Use the control panel application or the Windows Service Control Manager to start and stop services.

The following examples assume the current directory in the command line interpreter is changed to the location of the server executable file.

```
> owsas.exe -port 5020
```

This command runs the server in the background, thus as an invisible program. The server loads the default configuration file, if present, and listens at port 5020.

```
> owsas.exe -console -v -cfg d:\myowsas.cfg
```

This command runs the server with a console and verbose output enabled. The specified configuration file must exist.

```
> owsas.exe -v > owsas.log
```

The server is run in the background. The server loads the default configuration file, if present. Verbose output is written to the specified file.

```
> owsas.exe -h
```

The server displays the help screen and exits.

Stopping the Server

In Windows, the server can be stopped gracefully using the following methods:

- Stop the server in the control panel application.
- If the server has a console, press CTRL+C, press CTRL+BREAK, or click the console window's close button.

Linux

Required Privileges

Although not strictly necessary, the server should be run with root privileges. There are good reasons for this:

- The server has built-in drivers for a wide range of 1-Wire adapters. Some drivers require root privileges while other drivers don't, depending on permissions set in the file system of your particular Linux distribution. In most cases, root privileges are required.
- The 1-Wire Automation Software is installed as root user, hence all installed files are owned by the root user, including the provided configuration files. Most users want to use these configuration files, possibly modified, so the server must have root privileges in order to be able to access these files.

In conclusion, as a rule of thumb, run the server with root privileges.

Starting the Server

The following examples assume the current directory in the shell is changed to the location of the server executable file.

```
# owsas -port 5020 -v -cfg /home/peter/myowsas.cfg
```

The server listens at port 5020. Verbose output is enabled. The specified configuration file must exist.

```
# owsas -v > owsas.log &
```

This command runs the server in the background, detached from the console. The server loads the default configuration file, if present. Verbose output is written to the specified file.

```
# owsas -h
```

The server displays the help screen and exits.

If you want to run the server when Linux starts up, relying on **cron** is an good choice. You can edit the cron table as follows:

```
# crontab -e
```

Stopping the Server

In Linux, the server can be stopped gracefully using the following methods:

- Send signal SIGTERM to the server.
- If the server is attached to a console, press CTRL+C.

Use **kill** to send the SIGTERM signal. For example, if the process identifier of the server is 18104, send SIGTERM from the shell:

```
# kill 18104
```

Use **killall** if you want to specify the program name rather than the process identifier:

```
# killall owsas
# killall owsas-free
```

Note that **killall** sends the SIGTERM signal to all processes with the specified name. If you're running multiple instances of **owsas**, the **killall** command will terminate all of them.

5 Client Connections

Server Port

When the server starts up, the program creates a server socket port for accepting client connections over the network. The server can accept multiple client connections.

The port number can be specified in the server configuration file or on the command line using parameter **-port**.

Connection Limit

The maximum number of client connections can be limited in the configuration file by means of the **maxconn** keyword. For example:

```
maxconn 5;
```

The server accepts up to five incoming client connections. Any more connections will be refused.

Allowed IP Addresses

When keyword **allowip** occurs in the configuration file, the server will only accept incoming client connections from the specified list of IP addresses. For example:

```
allowip 192.168.1.105, localhost;
```

A client must be localhost (same computer as the server) or have IP address 192.168.1.105, else the server will refuse the connection.

User Authentication

Specify one or more **user** keywords in the configuration file to enable user authentication. Once enabled, all clients must authenticate. For example:

```
user "winston" "0new1reXL";
user "lizzy" "";
```

Each client that connects with the server must issue the **Authentication** command before any other command. For example:

```
auth "winston" "0new1reXL"
```

If the client passes a known username with the correct password, the server marks the connection as authenticated, if not the server immediately closes the connection.

Network Congestion

When the server sends responses to the client but the network can't keep up, network congestion is occurring and the server saves the responses in a large buffer for later transferring. However, if this buffer is full, the server drops any further responses until the network is transferring data again. The server ensures that entire response text lines are dropped, so the client won't receive partial responses.

6 Client Protocol

Once a client is connected to the server, it sends commands. The server returns responses. The set of commands and responses and related rules for transferring them is called the client protocol.

All commands and responses are encoded as UTF-8, fully supporting the Unicode character set. Note that the full range of Unicode characters only applies to strings containing descriptive text like adapter names, so it's perfectly possible to stick to ASCII characters by not using specific Unicode characters at all.

The client protocol suits both interactive communication (using a terminal program) and programmed communication (client software).

Command Parsing

The server parses each incoming command and checks for valid formatting. If a command is ill-formatted, it's discarded. If you want to track invalid commands, run the server with verbose printing turned on; the server will prints an error message for each incoming command that's ill-formatted.

Once an incoming command has been parsed successfully, it's ready for processing. Besides a few exceptions, most commands are added to the server's command queue.

Some commands are not queued but executed immediately. These commands are:

- **Cancel**.
- **Adapter Add**.
- **Authentication**.

You can specify multiple commands in the same line. Just separate the commands using a semi-colon:

```
dev "28-40CBBB2" add ; dev "28-40CBBB2" attr poll=on,60000
```

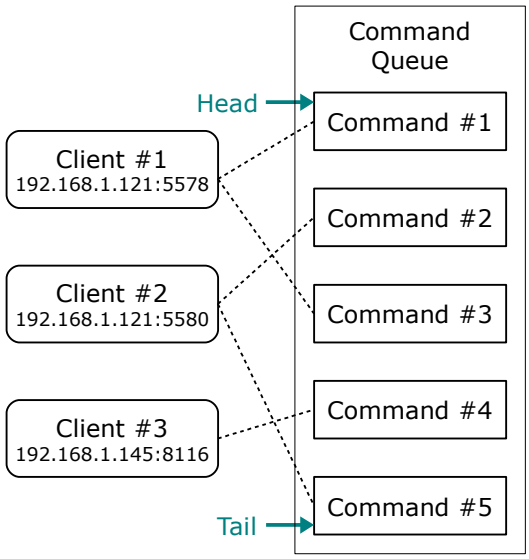
Command Queue

Commands sent by clients end up in the server's command queue. Commands are added to the tail of the queue in the order they come in.

All commands in the command queue are executed concurrently. Each command thus is a task in its own.

The server iterates through the command queue head to tail. As a result, commands are executed in the order they were queued, no matter how many iterations it takes to complete.

Commands that take just one iteration to execute are called synchronous commands. They will complete in the order they were



queued. For example, if the server receives these commands in the given order:

```
dev "1C-B8DD77F" add
dev "32-6DBDC0" add
```

then the commands will complete in the given order as they need just one iteration to fulfill their task.

When a command requires one or more iterations to execute, it's called an asynchronous command.

The concurrent activity of commands allows the server to perform lengthy operations at the same time, greatly improving performance. Operations like sensing data and the enumeration procedure benefit from this. For example, you can enumerate slaves on multiple adapters at once:

```
ch "ha7s-1":1:1 enum
ch "ha7s-2":1:1 enum
```

The responses of these commands are asynchronous as well. An asynchronous command returns its response (if any) when it's completed its task, and that moment is variable.

All asynchronous commands can be canceled. They are two variants of the cancel command:

```
cancel
cancel all
```

The first command cancels all asynchronous command that the client sent earlier. The latter command cancels all asynchronous commands of all clients.

Commands can be executed concurrently as long as they don't need to access the same master, slave or other resource. Doing so would introduce errors. The server enforces exclusive access to the same resource using an elaborate scheme of locking semantics. For example, the following commands are all asynchronous but will execute one at a time due to their accessing the same 1-Wire slave:

```
dev "3B-19CCEF" move ch 1:1:1
dev "3B-19CCEF" sense force
dev "3B-19CCEF" remove
```

When multiple commands access the same resource, the server guarantees that these commands are executed and completed in the order they were received. So the order in which the server grants access to the resource is deterministic rather than random.

The concurrent activity of commands may be undesirable in some situations. The server offers two client commands that control concurrency: **Block** and **Wait**.

Command **Block** prevents all subsequent commands in the queue from being executed; the **Block** command thus ends the iteration through the queue prematurely. As soon as all preceding commands have been completed, the **Block** command becomes the first command in the queue and also completes, thereby allowing subsequent commands to be executed.

A typical usage is shown in the following example:

```
hw enum
block
dump topo
```

The client wants to fully enumerate the adapters before dumping the topology. The **Block** command will unblock the **Dump Topology** command only after the enumeration commands have completed.

The Wait command is analogous to the Block command except that it completes after a given number of milliseconds. A typical usage involves enabling of an adapter followed by an enumeration:

```
adapter "abiowire" enable
wait 1000
adapter "abiowire" enum
```

Enabling the adapter is a synchronous command. Nevertheless it takes some time for the adapter to come up. The **Wait** command blocks the subsequent enumeration command for a while allowing the server to bring the adapter to a fully enabled state.

Macro Commands

A number of client commands do not directly end up in the command queue; they're split up into zero or more commands first. These so-called macro commands are always part of a base command that determines how the macro command will be split up.

Base commands are **Hardware**, **Adapter**, **Controller**, and **Channel**. Basically, the macro command is split up to the level of channels (**Probe**, **Enumerate**, **Untie**), controllers (**Detect**) or adapters (**Enable**, **Disable**).

The actual number of commands that end up in the command queue depends on the current composition (controllers and channels) and state (enabled or disabled) of the 1-Wire adapters.

Unsolicited Responses

If the server generates a response that doesn't originate from the completion of a command, it's called an unsolicited response.

Unsolicited responses are generated on a per-client basis, at the client's request, by means of the **Report** command. For example:

```
report add sensed dev "3A-000000052F6A"
report add sensed max31850
```

The client asks the server to report sensed data for the 1-Wire slave with ROM code 3A-000000052F6A and for all 1-Wire slaves that are based on a MAX31850 chip. Note that polling must be enabled for these devices; when a client issues an explicit **Device Sense** command, this will not lead to the generation of unsolicited responses.

Command Identifier

It's possible to assign an identifier to a client command. As a result, the server will always return a client response when the command has completed.

For example:

```
id 10 dev "20-14C3CF" add
id 10 done
```

Another example:

```
id 20 hw enum ; id 21 block ; id 22 dump
... enumeration responses ...
id 20 done
id 21 done
... dump of topology ...
id 22 done
```

The identifier is a 32-bit unsigned number. The client typically keeps track of the identifier and increments the value for each client command it sends to the server.

The use of command identifiers is particularly useful for client programs that need to track the lifespan of the client commands they send to the server. For example, the **1-Wire Automation Server GUI** program tracks command identifiers for animating the activity indicator and for highlighting command buttons like "ENUM".

Dump Command

The **Dump** command provides the client with a quick way to print out a block of useful information. Currently, these commands are defined:

`dump dn`	**Dump Device Nodes**
`dump topo`	**Dump Topology**
`dump`	Same as **Dump Topology**

Do not try to programmatically parse the contents of the information block. The formatting of the information is not officially defined nor is it guaranteed to remain unchanged in between software releases.

ROM Code Formatting Style

Various client commands and responses embed a 1-Wire ROM code. The client protocol defines two formatting styles for ROM codes: native and owfs.

A client may specify any formatting style in its client commands, the server understands all of them.

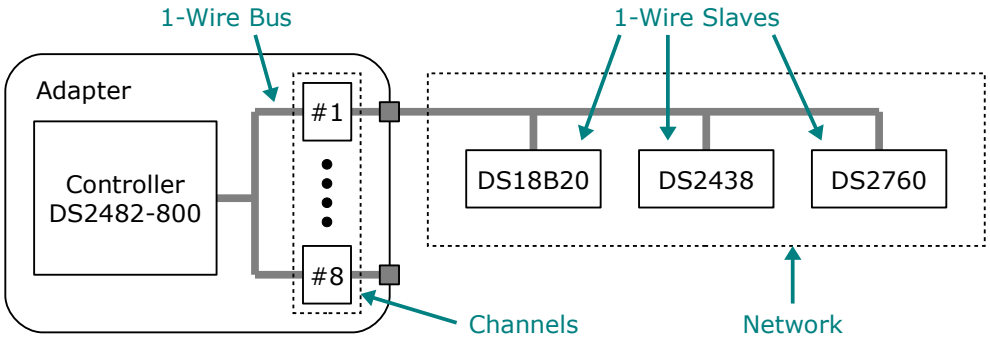
As for client responses, the server uses native as the default formatting style. Command line argument **-romcode** and configuration file keyword **romcode** can be specified to overrule the default setting. A client can change the current formatting style as follows:

`attr romcode=native`	Set native formatting style.
`attr romcode=owfs`	Set owfs formatting style.

The example commands and responses in this document use native formatting style. For all these example, owfs formatting style is equally applicable.

7 Hardware

Overview



This chapter explains the various concepts in 1-Wire hardware and the way the server works with these concepts.

Slave

A 1-Wire **slave** or **device** resides on a 1-Wire bus behind a 1-Wire channel. Each 1-Wire slave carries a unique 64-bit value called the ROM code. This value is world-unique meaning no two 1-Wire slaves in existence carry the same ROM code.

The server maintains information about each 1-Wire slave it knows and uses the ROM code as the unique identifier for looking up the device when needed.

There are several methods you can use to add 1-Wire slaves to the server:

- Run the enumeration procedure.
- Load a topology file.
- Use client command **Device Add**.

Many 1-Wire slaves have a built-in temperature sensor, ADC, counters, input lines and/or other sources that produce data. This data is called sensor data. The server assigns a sensor identifier to each slave that produces sensor data. Acquisition of sensor data is one of the main activities of the server.

A number of 1-Wire slaves provide programmable I/O (PIO) pins. The server reports the input state in the sensor data while offering a dedicated client command for controlling the output state of PIO pins.

Network

A **network** is a tree structure (or a hierarchical structure) of 1-Wire slaves. A network can hold multiple slaves in its root branch. 1-Wire hubs add sub-branches to the tree structure.

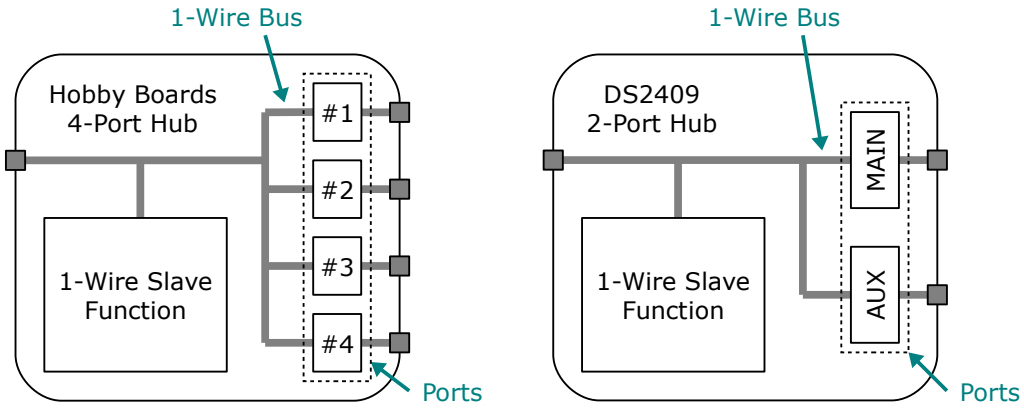
Hub

A 1-Wire **hub** is a 1-Wire slave that adds sub-branches to the 1-Wire network. A sub-

branch is hooked up to a **port** that provides the electrical characteristics required for the proper operation of a 1-Wire bus.

The server supports these hubs:

- Hobby Boards 4-Channel Hub. This device is a 1-Wire slave that adds four sub-branches to the 1-Wire network. The ports are numbered 1 to 4.
- Maxim DS2409 MicroLan coupler. This 1-Wire slave chip adds two sub-branches to the 1-Wire network. The first port is called MAIN, the second AUX.



When the server accesses a 1-Wire slave, it makes sure the hub ports that lay on the path from the controller to the target slave are activated first.

When you add hubs explicitly, you've to specify the type of hub in the **Device Add** client command:

```
dev "EF-15207BA3" add ds2409
dev "1F-56BA7" add hbh4
```

Channel

A 1-Wire **channel** is the starting point of a network of 1-Wire slaves which are physically connected to a 1-Wire bus. A channel is usually embodied as an RJ12 connector, an RJ45 connector, or a header.

Unlike USB or PCI, 1-Wire is not a plug-and-play protocol. This means the server must be explicitly informed about the whereabouts of the 1-Wire slaves. When the server is told that a slave is located in the network behind a channel, the server knows the slave can be accessed through that particular channel's 1-Wire master.

Maintaining the channels of the topology can be a tedious task. Luckily the server offers a number of powerful tools like the enumeration procedure, the detection procedure and topology files. Using these tools you should have no problem synchronizing the topology with your physical 1-Wire hardware.

Controller

A 1-Wire **controller** or **master** is a function that controls a 1-Wire bus. The controller divides the bus into one or more 1-Wire channels. Most controllers provide one channel. The DS2482-800 chip is an example of an 8-channel controller.

Note that a multichannel controller can only work with one channel at a time. A multichannel controller controls a single bus and as such it can communicate with one 1-Wire slave at any given time. The multichannel feature is there for electrical reasons, not for adding concurrent communication over multiple channels.

Adapter

A 1-Wire **adapter** represents a logical grouping of one or more 1-Wire controllers. Most adapters have one controller.

Some adapters comprise multiple controllers. For example, an AbioWire comes with two DS2482-800 controllers and one DS2482-100 controller for a total of three 1-Wire buses and seventeen 1-Wire channels.

Probing

Macro command **Device Probe** offers the functionality to probe for the presence of a specific 1-Wire slave on a physical 1-Wire bus. For example, lets see whether a DS2450 slave with ROM code 20-00000014C3CF-0E is present behind any of the available 1-Wire channels:

```
hw probe "20-00000014C3CF-0E"
probe ch "ow":1:1 "20-00000014C3CF" nonpresent
probe ch "usb-4-2":1:1 "20-00000014C3CF" present
probe done
```

Note that the probe command doesn't change the state of hub ports during probing. If hubs are present in the 1-Wire network and some ports are open, then the probe command will include those parts of the network that are visible to the 1-Wire master due to the open ports.

Device Nodes

The server works with a system of device nodes for defining adapters and controllers. The device nodes are organized in tree structure with a single root node. The device nodes provide a powerful tool for setting up 1-Wire adapters of many manufacturers in combination with a variety of common interface types like serial ports, USB-to-serial adapters, and the I2C bus.

You can dump the current state of the device nodes by sending the **Dump Device Nodes** client command. Example:

```
dump dn
Device Nodes:
+ ROOT
  + SERIAL: "/dev/ttyUSB0"
    + AXICAT
      + SERIAL: uart0
        + OWS-DRV: HA7S -> adapter "ha7s"
      + OWS-DRV: 1-Wire -> adapter "owaxicat"
  + I2CBUS: i2c-dev "/dev/i2c-1" [open:4]
    + OWS-DRV: DS2482 ad 18h [open] -> adapter "owhost"
    + OWS-DRV: DS2482 ad 19h [open] -> adapter "owhost"
```

```
    + OWS-DRV: DS2482 ad 1Ah [open] -> adapter "owhost"
    + OWS-DRV: DS2482 ad 1Bh [open] -> adapter "owhost"
  + OWS-DRV: DS2490 [open] -> adapter "usb-4-2"
```

A client uses commands **Adapter Add** and **Adapter Remove** to dynamically add and remove device nodes. As their names suggest, these commands works at the level of the 1-Wire adapter, hence they add or remove one or more controllers at once.

As stated earlier, an adapter comprises one or more controllers. From the viewpoint of the device nodes, all controllers must be located on the same hardware bus. This obviously is only possible if the hardware bus can host multiple slaves, like is the case with I2C. In the example, adapter "owhost" is composed of four I2C to 1-Wire controller chips all residing on the same I2C bus "/dev/i2c-1".

The LibUSB subsystem, if enabled, automatically adds device nodes when USB to 1-Wire adapters are plugged in. The DS2490 node in the dump above is an example of an automatically added device node.

Adapters can be enabled and disabled using client clients **Adapter Enable** and **Adapter Disable**. In the example, device nodes labeled "open" represent enabled controllers and their corresponding adapters.

Let's look at the client commands that are required to build the device nodes in the example:

```
adapter "ha7s" add serial "/dev/ttyUSB0" axicat uart0 ha7s
adapter "owaxicat" add serial "/dev/ttyUSB0" axicat ow
adapter "owaxicat" add i2cdev "/dev/i2c-1" ds2482 18h ds2482 19h ds2482
1Ah ds2482 1Bh
```

Remember that the last device node was automatically added by the LibUSB subsystem, so we don't have to issue an **Adapter Add** command for adding this device node.

The nodes from the root node to a controller's device node make up a so-called **device path**. Each controller has a unique device path.

The parameters specified in the **Adapter Add** command basically describe one or more device paths, one for each controller.

Each adapter must have a unique name consisting of 1..16 Unicode characters. If the client doesn't specify a name in the **Adapter Add** command, the server assigns a unique name. Its possible to change an adapter's name:

```
adapter 2 attr name="100€ adapter"
adapter "usb-4-10" attr name="ff32-1"
```

LibUSB Subsystem

The server incorporates a component, the LibUSB subsystem, that tracks plug-and-play events of USB devices.

When the LibUSB subsystem is enabled, it tracks the arrival of supported USB to 1-Wire adapters. When such an adapter is plugged in, the server creates a device node and enables the adapter.

When the LibUSB subsystem is disabled, the arrival of USB devices is neglected. This doesn't mean that connected USB to 1-Wire adapters cease to function! As long as one or more connected adapters stay plugged in, the corresponding device nodes remain in

existence and you can enable, disable and remove the adapters.

When the server starts up, the LibUSB subsystem is disabled by default. Once the server is up and running, you can issue the following client commands to enable and disable the LibUSB subsystem:

`lu enable`	**LibUSB Enable**
`lu disable`	**LibUSB Disable**

The LibUSB subsystem supports the following USB devices:

Vendor ID	Product ID	Name
04FAh	2490h	Maxim DS2490. This chip can be found in adapters like the DS9490R and DS9490B.
04D8h	F8B9h	Flyfish FF32 and FF34. These devices shares the same USB IDs. The server determines the model when the 1-Wire adapter is enabled.
0DE7h	0191h	USBMicro U401.
0DE7h	01A5h	USBMicro U421.
0DE7h	01C3h	USBMicro U451.

The LibUSB subsystem depends on the presence of the LibUSB library. The server loads the library as soon as the LibUSB subsystem is enabled. If the library can't be found, the server reverts the LibUSB subsystem to the disabled state.

When the LibUSB subsystem is disabled, the library is kept in memory as long as associated USB to 1-Wire adapters are present. Once all associated USB to 1-Wire adapters are gone, the library is unloaded.

W1 Connector

The W1 subsystem, part of the Linux kernel, provides a framework for managing 1-Wire masters and 1-Wire slaves. The W1 subsystem offers a connection to applications. The 1-Wire server incorporates a component, called W1 connector, that connects to the W1 subsystem.

When the server starts up, W1 connector is disabled by default. Once the server is up and running, you can issue the following client commands to enable and disable the W1 connector:

`w1 enable`	**W1 Enable**
`w1 disable`	**W1 Disable**

When the W1 connector is disabled, the server disconnects from the Wire1 interface and removes all associated 1-Wire adapters.

There are drawbacks when using W1. First, the W1 subsystem is slow compared to the native support offered by the 1-Wire server.

Secondly, the W1 subsystem regularly enumerates its 1-Wire networks in the background without synchronizing with commands coming from the server. When the enumeration takes off in the middle of a command sequence, errors or timeouts will occur. Note that

the server ignores the background enumeration completely; use the enumeration procedure for discovering your 1-Wire slaves.

W1 doesn't support probing of 1-Wire slave nor the detection procedure.

You typically use W1 with 1-Wire masters that are not supported by the server. For example, you can connect a DS18B20 temperature sensor to the GPIO pins of a Raspberry Pi, start the w1-gpio kernel driver, and enable the W1 connector to access the temperature sensor.

8 Topology

The server's view on the 1-Wire slaves is called the **topology**. The topology contains the 1-Wire slaves that the server knows about.

The topology is subdivided in three areas: channels, unallocated channels and non-present devices.

Client command **Dump Topology** prints out the current topology. Example:

```
dump topo
Non-present devices:
  30-000012B5735B-F4  DS2760/2761/2762 Li+ battery monitor
  3B-00000019CCEF-07  DS1825 thermometer
Network 1:
  1C-00000B8DD77F-C8  DS28E04-100 4Kb EEPROM with PIO
  32-0000006BDBC0-DB  DS2780 standalone fuel gauge
  36-000003612CDB-84  DS2740 Coulomb counter
  3B-0000001529B5-FC  MAX31826 thermometer
Network 2:
  35-00000043F47C-E9  DS2755/2756 high-precision battery fuel gauge
  01-000016707B5C-40  DS2401/2411/1990A silicon serial number
Network 3:
  3D-00000081D206-15  DS2775/2776/2781 Li+ fuel gauge
  3B-000000183368-2E  MAX31850 thermocouple
01: USB to 1-Wire "usb-4-3"
  01: DS2490
    01: 28-0000040CBBB2-C4  DS18B20 thermometer
    01: 42-00000038D0BE-A3  DS28EA00 thermometer
    01: 81-000000324BBD-31  DS1420 serial ID button
    01: 29-00000011BD2A-4E  DS2408 8-channel addressable switch
02: AbioWire "aw"
  01: DS2482-800
    01: 22-0000003201DA-1C  DS1822 Econo digital thermometer
    01: 3A-000000052F6A-85  DS2413 dual-channel addressable switch
    01: 01-000016707B5B-C5  DS2401/2411/1990A silicon serial number
  02: DS2482-800
    01: 20-00000014C3CF-0E  DS2450 quad A/D converter
    01: 10-000802A49A17-B1  DS18S20/1920 thermometer
  03: DS2482-100
```

Several commands allow a client to manipulate the topology:

- Run the enumeration procedure.
- Run the detection procedure.
- Load a topology file.
- Add, remove, and move a device.
- Add, remove, move, and clear a device group.
- Add and remove an unallocated channel.
- Add, remove, enable, and disable an adapter.

Channels

The channels are the place where the topology meets the physical hardware. A channel is the starting point of a network of 1-Wire slaves which are physically connected to a 1-Wire bus.

Unallocated Channels

An unallocated channel stores a 1-Wire network that's not currently connected to any channel.

Unallocated channels are most often used in conjunction with the detection procedure and the removal and disabling of adapters.

Non-present Devices

The non-present devices (NPD) area in the topology stores information about 1-Wire slaves that are not part of a channel or unallocated channel. This way information about 1-Wire slaves can be retained.

There's no hierarchy in the NPD, just the root level. All slaves including hubs that are moved or added to the NPD reside at the root level. This means that when a hub is moved from a channel or unallocated channel to the NPD, all slaves that are located behind ports are disassociated from the hub and end up at the same level in the NPD.

The NPD area is most often used in the context of the enumeration procedure. When the server enumerates a channel, it first moves all present 1-Wire slaves to the NPD in order to empty the channel.

The NPD comes in handy when you want to configure a 1-Wire slave for polling before it's actually known to the server. For example:

```
dev "1C-B8DD77F" add
dev "1C-B8DD77F" attr poll=on,60000
```

In this example, the slave is first added. If it's not known to the server, it's added to the NPD, else it stays put wherever it is located in the topology. Next, polling is turned on. As soon as the slave moves to a position in a network behind a 1-Wire channel, the server starts polling the slave.

Enumeration Procedure

The enumeration procedure is a very powerful tool for discovering 1-Wire slaves that are present in a physical 1-Wire network. Client command **Enumerate** initiates the enumeration. It's a macro command that operates at the level of the 1-Wire channel.

When the server enumerates a channel, it first moves all present 1-Wire slaves to the NPD in order to empty the channel. Then discovery of 1-Wire slaves begins. During the discovery process, 1-Wire slaves may be moved between the NPD and the channel several times, especially when hubs are present in the 1-Wire network.

Example enumeration commands:

```
hw enum
adapter "ff32" enum
ctrl "abiowire":2 enum
ch 1:1:1 enum
```

Features of the enumeration procedure include:

- Support for the Hobby Boards 4-Channel Hub.
- Support for the DS2409 MicroLan Coupler chip, a 2-port hub.
- Usage of DS2409 Smart-ON feature for faster enumeration.
- Discovery of arbitrarily deep networks (nested hubs).
- Support for hardware-accelerated enumeration:
  - AxiCat 1-Wire Master.
  - OW-SERVER-ENET-2.
  - HA7E/HA7S.
- Recognition of type of family 3Bh chip (DS1825, MAX31826, MAX31850).
- Recognition of Hobby Boards devices.

The enumeration procedure is a manual tool. There's no reason for the server to automatically perform enumerations. You've to see enumeration as a tool for adding your physical 1-Wire networks to the server's topology. Once you've set up your 1-Wire networks, they usually remain static, so there's little reason for the server to run the enumeration procedure periodically.

You can set an adapter to automatically enumerate upon arrival. For example:

```
adapter "enet" add enet "192.168.1.101" 8080
adapter "enet" attr enum=on
adapter "enet" enable
```

Although it's preferred to turn on detection, some 1-Wire adapter like the OW-SERVER-ENET-2 are substantially faster doing enumeration than detection. Note that when the OW-SERVER-ENET-2 disappears and reappears, due to the nature of the enumeration procedure an empty unallocated channel will remain. If this occurs regularly, empty unallocated channels will pile up. To avoid this situation, you're advised to issue client command **UCH Purge** periodically.

Detection Procedure

The aim of the detection procedure is find out where networks of unallocated channels are physically located behind 1-Wire channels.

During detection, the server picks 1-Wire slaves from the unallocated channel's network and probes for their presence behind the channels of the controller. The 1-Wire slaves are picked from the root level of the network. They're picked not just randomly, but in a strict order based on their family code:

1. DS2401/2411/1990A Silicon Serial Number chips (family code 01h). These slaves have the specific purpose of identifying a 1-Wire network.
2. DS1420 Serial ID Button chips (family code 81h). These slaves are usually part of DS9490 adapters.
3. The remaining slaves, a.k.a. non-identification slaves. The server only uses a limited number of these slaves. The default is one slave.

As soon as one slave has been successfully detected behind a channel, the server moves the entire network from the unallocated channel to the channel and removes the unallocated channel from the topology.

You can change the number of non-identification slaves to use for each unallocated channel, like this:

```
uch 1 attr nid=10
```

The number has a valid range of 0..255. Zero indicates the server must not use any non-identification slaves.

The detection procedure works at the level of the 1-Wire controller. All channels of the controller are involved during detection. Nonetheless, it's possible to narrow down the procedure to a single channel on a multichannel controller. For example, let's run the detection procedure on the fifth channel of the second DS2482-800 controller on an AbioWire:

```
ch "abiowire":2:5 detect
```

In combination with a topology file, the detection procedure can effectively provide an efficient replacement for enumeration. The following commands are typically sent when the server starts up:

```
topo load "abiowire_networks.txt" force
adapter "abiowire" add i2cdev "/dev/i2c-1" abiowire
adapter "abiowire" attr detect=on
adapter "abiowire" enable
```

The commands instruct the server to load a topology file containing networks that are to be detected on the AbioWire. The AbioWire is added, set to automatic detection, and enabled. As soon as the AbioWire is fully enabled, the server commences the detection procedure on all three 1-Wire controllers in search of the networks that the topology file has provided.

The detection procedure is also useful for recovering from a situation where a 1-Wire adapter may disappear and reappear again due to the nature of its physical and/or mechanical connection. A good example is a DS9097U adapter connected to a USB-to-serial adapter. Although the DS9097U is firmly connected to the USB-to-serial adapter, the USB-to-serial adapter itself may be unplugged and replugged, or it may disappear from the USB bus and come back due to electromagnetic interference for example. Here's an example how to recover:

```
topo load "networks.txt" force
adapter "ow" add serial "/dev/ttyUSB0" ds9097u
adapter "ow" attr reconn=on,5000 detect=on
adapter "ow" enable
```

When the USB-to-serial adapter is unplugged, the DS9097U adapter will be brought to the disabled state and the server will move the 1-Wire network to a newly created

unallocated channel. When the USB-to-serial adapter is plugged in a bit later, the server will be able to re-enable the DS9097U. As soon as the DS9097U is enabled again, the detection procedure kicks in and the network is moved from the unallocated channel to the DS9097U's channel (assuming the physical network hasn't changed).

It's possible to turn on automatic detection by default. This feature is especially useful when using USB-based adapters that are automatically added (DS2490, FF32, FF34):

```
hw attr detect=on
```

Technically, if the detection attribute for an adapter is set to "on" or "off", it overrules the detection attribute for all hardware. When an adapter is added either automatically (USB) or by issuing client command **Adapter Add**, its detection attribute is set to "default" meaning the adapter uses the detection attribute for all hardware.

Untying

The opposite of detection is called untying. This happens when a network is moved from a channel to a new unallocated channel.

The following events result in the untying of networks:

- An adapter disappears from the system. For example, when you unplug a DS9490 adapter.
- The client issues command **Adapter Disable**.
- The client issues command **Adapter Remove**.
- The client issues command **Untie**.

Client command Untie comes in handy when you move one or more physical networks from one channel to another, even between different adapters. After you've moved the physical networks, you can untie the affected channels in the topology and run the detection procedure to let the server figure out where the networks have moved to.

For example, suppose you physically unplug all networks from an OW-SERVER-ENET-2 adapter named "enet" and connect them to an AbioWire named "aw". After you're done with the changes, you can update the server's topology with these client commands:

```
adapter "enet" untie
adapter "aw" detect
```

Just make sure you move your physical networks as a whole, not parts of it. If you change the structure of a network, you better run the enumeration procedure to rebuild the topology.

9 Sensor Data

Overview

Acquisition of **sensor data** from 1-Wire slaves is one of the main activities of the server. The server supports many 1-Wire slaves including temperature sensors, A/D converters, battery monitors, counters, and PIO chips.

For each known 1-Wire slave, the server keeps data you should be aware of:

- Sensor identifier: Indicates the actual sensor information that the 1-Wire slave produces.
- Current sensor data: The sensor data that was acquired most recently. This data is marked unavailable when the 1-Wire slave hasn't been read yet. If marked as unavailable, the data is marked valid or invalid, depending on whether the slave was successfully read or not.
- Previous sensor data: A copy of the sensor data that was acquired before the current sensor data. By keeping this data the server can keep track of differences in sensor data.
- Polling information: Polling interval (milliseconds) and is-polling-enabled state.

The server also keeps information per client connection:

- Temperature scale: The scale used when formatting the sensor data string. The scale can be set to Celsius (default), Fahrenheit, or Kelvin.
- Reporting: A list of 1-Wire slave ROM codes and a list of sensor identifiers that trigger the generation of unsolicited responses.

Sensing

Client command **Device Sense** allows a client to acquire the sensor data of a 1-Wire slave. For example, let's read a temperature sensor:

```
dev "28-0000040CBBB2-C4" sense force
dev "28-0000040CBBB2" sensed ds18b20 "2014-11-12 02:44:47 0132 +019.1 C
parasite"
```

Label “force” tells the server to always read the 1-Wire slave even when previously acquired sensor data is stored.

If you omit label “force”, the server will return the stored sensor data rather than actually accessing the 1-Wire slave, unless there's no sensor data available in which case label “force” has no effect and the server accesses the 1-Wire slave. The latter usually happens when the 1-Wire slave is sensed for the first time.

The response includes the ROM code of the 1-Wire slave, sensor identifier, and the **sensor data string**. The string always starts with the date and time when the sensor data was acquired, followed by the actual sensor data.

The sensor data string shown in the example contains the value of the temperature register and a decimal representation of the temperature in Celsius. The representation is derived from the register value and will usually do for your purposes. The register value is there in case you need the highest precision.

Whenever the server returns a temperature value, the server uses a scale for the representation. The available scales are Celsius, Fahrenheit and Kelvin. The scale is configured per client connection, the default is Celsius. Use client command **Attribute** to change the temperature scale.

If the client tries to sense a 1-Wire slave that's not located behind a channel or that's not part of the topology at all, the server responds as follows:

```
dev "28-0000040CBBB2-C4" sense
dev "28-0000040CBBB2" sensed nonpresent
```

If the 1-Wire slave does reside behind a channel, but it can't be accessed over the 1-Wire bus, the server responds as follows:

```
dev "28-0000040CBBB2-C4" sense force
dev "28-0000040CBBB2" sensed ds18b20 "2014-11-12 02:44:47 ERROR"
```

The presence of a sensor data string indicates the server really has tried accessing the 1-Wire slave. The "ERROR" label says that access has failed. Usually this means the 1-Wire slave is disconnected from the 1-Wire bus or the 1-Wire slave is located behind the wrong channel or hub port in the topology.

Short Responses

You can also have the server return so-called short responses. A short response consists of a single value. For example:

```
dev "28-40CBBB2" sense force rsp=temp
19.5
```

This command just returns the temperature value. Note that the value is formatted a bit differently than the representation in the sensor data string; leading zeroes and the plus sign are omitted. The idea is that a client uses short responses when simple processing of sensor data is at hand.

Label "temp" is called the short response identifier. For each sensor identifier, the server defines a set of response identifiers. Some 1-Wire slaves have an elaborate set of short response identifiers, like the MAX31850.

It's important to know that a forced sense command will acquire all sensor data, not just the one specified in the short response. This attribute of command **Device Sense** allows a client to read a 1-Wire slave and request the sensor values in shorthand form like this:

```
dev "42-38D0BE" sense rsp=sensedt all force
dev "42-38D0BE" sense rsp=temp
dev "42-38D0BE" sense rsp=piosensed
dev "42-38D0BE" sense rsp=pioset
dev "42-38D0BE" sense rsp=pwrmode
2014-12-10 15:20:24
19.4
3
3
external
```

If the result of the requested short response isn't available, the server returns "ERROR". For example:

```
dev "42-38D0BE" sense rsp=pwrmode
ERROR
```

A set of short responses is available for all sensor identifiers:

Name	Return Values	Description
valid	**0** **1**	One if sensor data is valid, zero if sensor data is invalid or unavailable.
sensorid	Example: **ds2408** **ERROR**	The sensor identifier of the 1-Wire slave.
sensedt all	Example: **2014-12-10 15:20:24** **ERROR**	Date and time the sensor data was acquired.
sensedt year	**0..9999** **ERROR**	The year the sensor data was acquired.
sensedt month	**1..12** **ERROR**	The month the sensor data was acquired.
sensedt day	**1..31** **ERROR**	The day the sensor data was acquired.
sensedt hour	**0..23** **ERROR**	The hour the sensor data was acquired.
sensedt minute	**0..59** **ERROR**	The minute the sensor data was acquired.
sensedt second	**0..59** **ERROR**	The second the sensor data was acquired.

Polling and Reporting

You can instruct the server to poll a 1-Wire slave periodically and report the sensor data in unsolicited responses. This takes two commands:

- Issue client command **Report Sensed** to enable unsolicited responses.
- Issue client command **Device Attribute Poll** for the 1-Wire slave.

For example, let the server poll our DS18B20 temperature sensor each minute:

```
report add sensed dev "28-40CBBB2"
dev "28-40CBBB2" attr poll=on,60000
```

You can freely choose the order of the commands, the results are the same, but there's a catch. When you issue command **Device Attribute Poll**, the server immediately starts polling the 1-Wire slave. So if you issue command **Report Sensed** some time later, you may miss the first polled sensor data.

The client can instruct the server to report sensor data only when it has changed. For example:

```
report add sensed dev "28-40CBBB2" diff
```

This example command tells the server that the client wishes to receives sensor data when it differs from the previously acquired sensor data.

Take note that client command **Report Sensed** operates at the level of the individual client connection, while client command **Device Attribute Poll** takes effect at the server level thus affecting all clients. This means multiple client can enable unsolicited

responses for the same 1-Wire slave or family, while it takes just one command to enable polling of the 1-Wire slave or family.

Temperature Scale

When the server returns a temperature value, either in the sensor data string or in a short response, it uses a temperature scale. By default, the scale is set to Celsius when the client connects to the server.

Use client command **Attribute** to change the temperature scale for the client connection.

`attr tscale=celsius`	Set temperature scale to Celsius.
`attr tscale=fahrenheit`	Set temperature scale to Fahrenheit.
`attr tscale=kelvin`	Set temperature scale to Kelvin.

10 Programmable I/O Pins

Overview

A number of 1-Wire slaves provide programmable I/O (PIO) pins. These pins act as general-purpose digital I/O lines and can be remotely controlled over a 1-Wire network.

The server reports the input state and activity state of PIO pins in the sensor data string of the **Device Sensed** client response. For controlling the output state of PIO pins the server offers a dedicated client command called **Device PIO**.

Reading State of PIO Pins

In the realm of 1-Wire, a PIO pin can have three states:

- Sensed state: The state on sampled the pin, a.k.a. the input state.
- Output state: The state written to the pin.
- Activity state: Indicates whether the input or output state of the pin has changed.

Not all slaves implement the activity state.

There's no explicit setting of the input/output direction. See the datasheet of your particular 1-Wire slave for more details on how to cope with direction.

For example, the DS2408 slave implements all three states:

```
dev "29-00000011BD2A-4E" sense force
dev "29-00000011BD2A" sensed ds2408 "2014-11-12 02:37:51 01000000 11111110
01100001 reset  external"
```

The sensor data in the example shows that the input state of 7th pin is one, all pins but the 1st have output state one, and three pins have their activity bit set.

Controlling Output State of PIO Pins

Client command **Device PIO** controls the output state of PIO pins. This command works for all supported 1-Wire slaves that incorporate one or more PIO pins.

Let's change the output state of some pins on a DS2408 slave:

```
dev "29-00000011BD2A" pio off 1 on 4 5
```

The example command clears the output state of the 1st pin, and sets the output state of the 4th and 5th pin. PIO pins are numbered from 1 onwards.

You can also specify a bit mask of pins:

```
dev "29-00000011BD2A" pio set 80h clear 110b
```

This command set the output state of the 8th pin, and clears the output state of the 2nd and 3rd pin.

It's perfectly possibly to combine arguments “on”, “off”, “set” and “clear”. The server accumulates all arguments into two pin masks, a mask for setting pins, and a mask for clearing pins. If the two masks overlap for some pins, then these pins are set (setting takes precedence over clearing). Unspecified PIO pins remain untouched.

11 Configuration File

Overview

When the server starts up, it optionally processes a configuration file. This file contains a number of settings that aren't strictly necessary but can't be set elsewhere.

The server treats the configuration file as read-only. The server will never write a configuration file to disk.

The configuration file must use one of these character encoding schemes: UTF-8, UTF-16 Little Endian, or UTF-16 Big Endian. The server recognizes the character encoding by means of the byte order mark (BOM) at the beginning of the file. If no byte order mark is present, the server assumes the file is encoded as UTF-8.

Note that UTF-8 encoding is a super set of ASCII characters 0..127. This means you can create a configuration file with a simple ASCII editor.

Path

If command line parameter **-cfg** is specified, the given filename overrules the default configuration file. The filename may include an absolute or relative path. If the path is relative, the location of the server's executable file acts as the base directory. The given filename must exist, else the server stops with an error.

If command line parameter **-cfg** isn't specified, the server assumes a default configuration file is located in the program directory. The default configuration file is called **owsas-free.cfg** for the free version of the server and **owsas.cfg** for the full version of the server.

The presence of the default configuration file is optional, the server can run without it. Note that in case the default configuration file is not present, then at least command line parameter **-port** must be specified, else the server doesn't know what port to listen at.

Contents

The configuration file consists of keywords followed by one or more parameters.

Keyword	Description
`port`	The number of the server port the server listens at.
`maxconn`	The maximum number of client connections allowed.
`allowedip`[1]	A list of allowed IP addresses clients can connect from.
`user`[1]	A user account.
`romcode`	Default ROM code formatting style for client connections.
`cmdfile`[1]	File containing client commands to be executed when the server starts up.

[1] Not available in the free version.

Port

You can specify the server port in the configuration file. If you don't, you'll have to specify the server port with the **-port** command line parameter, else the server doesn't know which port to listen at.

Example:

```
port 5001;
```

Valid port numbers are 1..65535.

If this keyword occurs multiple times in the configuration file, the last one will take effect.

Connection Limit

Keyword **maxconn** sets the maximum number of incoming client connections. You can specify a non-zero value to set a limit, or value zero to allow an unlimited number of connections.

```
maxconn 0;
```
Allow unlimited number of client connections.

```
maxconn 10;
```
Allow up to 10 client connections.

If this keyword occurs multiple times in the configuration file, the last one will take effect.

Allowed IP Addresses

You can specify a list of IP addresses from which clients are allowed to connect. The list is a comma-separated enumeration of IP addresses.

Example:

```
allowip
    192.168.1.105,
    192.168.1.110,
    localhost;
```

The label "localhost" is equivalent to "127.0.0.1" and refers to the system the server is running on.

In the absence of a list of allowed IP addresses, the server will accept incoming client connections.

This keyword can be specified multiple times in the configuration file. The server adds the IP addresses to a single list.

User Authentication

Add one or more users to enable authentication. The parameters of the user keyword are a user name followed by a password. The password is optional.

```
user "winston" "0new1reXL";
user "lizzy" "";
```

When one or more users are added, the server expects each incoming client connection to authenticate itself by issuing client command **Authentication** as the first command.

ROM Code Formatting Style

Keyword **romcode** selects the formatting style of ROM codes in client responses.

`romcode native;`	Native formatting of the server.
`romcode owfs;`	Formatting of the owfs software.

If this keyword occurs multiple times in the configuration file, the last one will take effect.

Command line parameter **-romcode** overrules this keyword. If neither this keyword nor the command line parameter is specified, the server uses **native** formatting. Individual client connections can send command **Attr** to select a different formatting style.

Command File

The configuration file can point to a file that contains client commands that are to be executed when the server starts up. Note that any responses that result from executing these client commands will be discarded.

For example:

```
cmdfile "owsas-startup-cmds.txt";
```

This keyword can occur only once in the configuration file.

IMPORTANT! The command file is processed in the context of a separate client connection. If you've set up one or more users in the configuration file, the command file must start with an **Authentication** client command. If authentication fails, the server won't process the command file any further.

12 Topology Files

Overview

A topology file contains information about 1-Wire slaves and the structure of 1-Wire networks. Topology files are primarily used in combination with the detection procedure.

A topology file is a human-readable Unicode text file and can be manually edited. The format allows for future extensions and is upwards and backwards compatible.

Saving

Client command **Topology Save** saves the server's current topology to a topology file. It's like taking a snapshot of the topology. The command is synchronous.

Saving a topology file involves these steps:

1. The server saves all 1-Wire slaves in the NPD.
2. The server saves each non-empty unallocated channel as a network.
3. The server saves each non-empty channel as a network.

Example client command:

```
topo save "mytopology.txt" utf8
```

You can explicitly choose the character encoding of the output file: **utf8** (UTF-8), **utf16le** (UTF-16 Little Endian), or **utf16be** (UTF-16 Big Endian). If the encoding is omitted, the server chooses a default one depending on the system it's on (UTF-8 for Linux, UTF-16 LE for Windows).

The server will overwrite the target file if it already exists.

Loading

Client command **Topology Load** instructs the server to load a topology file. The command is asynchronous, it can takes a while to complete and it can be cancelled.

Examples:

```
topo load "mytopology.txt"
```

The server loads the given topology file. 1-Wire slaves and unallocated channels are added as the server reads the information from the file. When the server reads information about a 1-Wire slave that already exists in the topology (same ROM code), the information is discarded and the existing 1-Wire slave remains unchanged.

```
topo load "mytopology.txt" force
```

This command changes the behavior of the server when it reads information about a 1-Wire slave that exists in the topology. With the **force** flag in place, the server will remove the 1-Wire slave and recreate a new one in the NPD based on the information read from the topology file.

Path

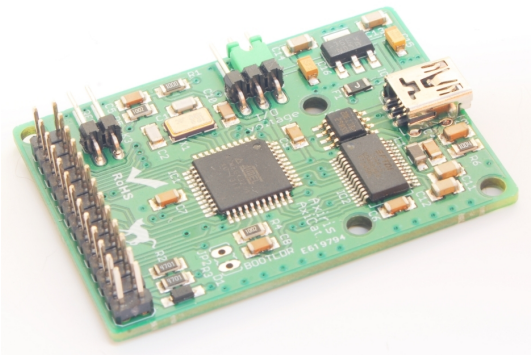
Topology files are stored on the host system of the server, not on the client side. When you specify a topology file name and optionally a path, it's important to follow the syntax rules of the file system on the server's host system.

You can specify a path with the topology file name. If the path is relative, the location of the server's executable file acts as the base directory.

13 1-Wire Masters

AxiCat Adapter

The AxiCat is a USB-I2C/SPI/1-Wire/UART/GPIO interface adapter.



The server support the following interfaces: 1-Wire master, I2C master, UART0, and UART1. The interfaces operate concurrently, making the AxiCat a powerful adapter that can interface with multiple 1-Wire masters and 1-Wire slaves at once.

The voltage level of the I/O interface can be set to 3.3 V or 5 V. This enables the use of 3.3 V 1-Wire slaves like the MAX31850 thermocouple.

Although the AxiCat technically is a USB to FIFO device, the operating system installs the AxiCat as a USB-to-serial port adapter. As such, label "serial" must occur in client command **Adapter Add** and you've to specify the serial path of your AxiCat.

UART0 and UART1

You can connect a serial to 1-Wire adapter that operates at 3.3V or 5 V to the AxiCat. For example, let's hook up a HA7S to interface UART1:

```
adapter "ow" add serial "/dev/ttyUSB0" axicat uart1 ha7s
```

Since the HA7S is designed to operate at 5 V, make sure the AxiCat is set to 5 V as well.

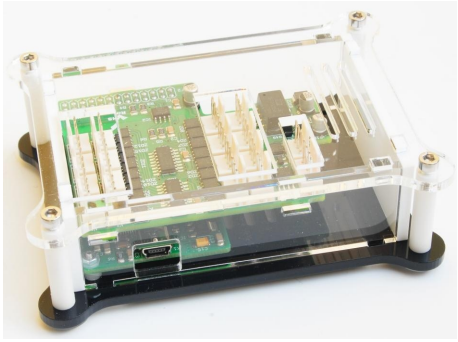
I2C Master

The I2C master can interface with all supported I2C to 1-Wire controllers. For example:

```
adapter add serial "/dev/ttyUSB0" axicat twi ds2482 1Eh ds2482 1Fh
```

You can plug an AbioWire onto an AxiCat and let the server use the combo. For example:

```
adapter "abiowire" add serial "/dev/ttyUSB0"
axicat twi abiowire
adapter "abiowire" attr reconn=on,5000
adapter "abiowire" enable
```



These three commands turn the AbioWire into a true plug-and-play USB device.

IMPORTANT! The AbioWire is a 3.3 V device. Make sure the jumper (JP1) on the AxiCat is set to 3.3 V operation.

1-Wire Master

The AxiCat implements a 1-Wire master with accelerated enumeration and probing of 1-Wire slaves and strong pull-up control. Here's an example how to add the 1-Wire master to the server:

```
adapter "ow" add serial "\\.\COM14" axicat ow
```

DS2480B Serial to 1-Wire Controller

The DS2480B chip bridges between a serial interface and a 1-Wire bus. It can be found in a number of adapters like the DS9097U.

```
adapter add serial "\\.\COM1" ds2480
```

Label "ds2480" tells the server to use the DS2480B driver.

DS9097U Adapter

This adapter uses the DS2480 Serial to 1-Wire controller chip. It can be directly connected to an RS-232 serial port.

The adapter draws power from the serial port's DTR and RTS lines. If neither of these lines are activated, the adapter won't function. The server must be told to activate these lines:

```
adapter add serial "\\.\COM1" ds9097u
```

Label "ds9097u" tells the server to use the DS2480B driver and activate DTR and RTS.

HA7E/HA7S Adapter

EDS produces two Serial to 1-Wire adapters called HA7E and HA7S. Functionally, these adapters are the same. Electrically, the HA7E interfaces with a standard serial port while the HA7S requires a 5 V level serial interface.

Client command **Adapter Add** accepts labels "ha7e" and "ha7s" for adding an device node for these adapters. The labels mean the same to the server, so you can specify "ha7s" for a HA7E and the other way around. The reason both labels exists is that client command **Dump Device Nodes** can show an appropriate description of the adapter.

Examples:

```
adapter "ha7s" add serial "\\.\COM14" ha7e
```

The command adds device nodes for a HA7E that's connected to a USB-to-serial adapter. The system is running Windows.

```
adapter "ha7e" add serial "/dev/ttyUSB0" axicat uart1 ha7s
```

The command adds device nodes for a HA7S that hook up to an AxiCat that's plugged into a USB port on a Linux system.

DS9097 Adapter

The DS9097 is a passive adapter that performs level conversion between RS-232 and 1-Wire. This adapter relies on the UART for generating and sampling the 1-Wire signals.

The DS9097 is a simple circuit that sits between an RS-232 serial port and the 1-Wire network. It's primarily used with on-board UARTs but also performs well when interfacing with a USB-to-serial adapter.

The DS9097 adapter requires a DTR line to function properly. It doesn't support strong pull-up.

Various implementations of passive adapters are available. They should work well with the server provided they're compatible with the DS9097.

Example:

```
adapter add serial "/dev/ttyS0" ds9097
```

Label "ds9097" tells the server to use the DS9097 driver and activate DTR.

DS2482 I2C to 1-Wire Controller

The DS2482 chip is a 1-Wire master that incorporates a I2C slave function for interfacing with a host system. It's found on various adapters and boards. There are two variants of this chip:

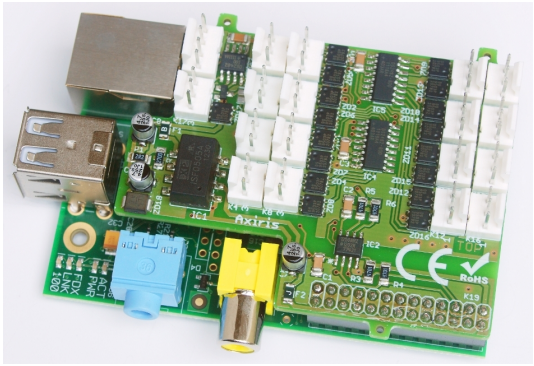
- DS2482-100: A single-channel 1-Wire master.
- DS2482-800: An eight-channel 1-Wire master.

The server can distinguish between a DS2482-100 and a DS2482-800.

```
adapter add i2cdev "/dev/i2c-2" ds2482 20h
```

Label "ds2482" is followed by the I2C slave address assigned to the DS2482 chip.

AbioWire 1-Wire Adapter



The AbioWire was initially developed for use with the Raspberry Pi A and B single board computers. Using the USB-based AxiCat adapter, you can connect the AbioWire to a multitude of computer systems.

The AbioWire comprises three I2C to 1-Wire controllers for a total of seventeen channels:

- DS2482-800 with slave address 18h.
- DS2482-800 with slave address 19h.
- DS2482-100 with slave address 1Ah.

The adapter also features a battery-backed PCF2129A real-time clock chip.

Let's look at examples of an AbioWire fitted on a Raspberry Pi:

```
adapter "abiowire" add bscdetect abiowire
```

The command tells the server to detect how to interface with the AbioWire.

```
adapter "abiowire" add bsc1 abiowire
```

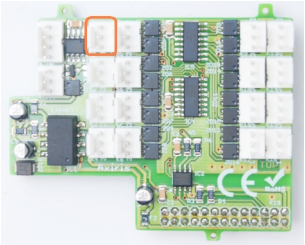
The server uses direct I/O with BSC 1 to interface with the AbioWire.

```
adapter "abiowire" add i2cdev "/dev/i2c-1" abiowire
```

The server uses the i2c-dev subsystem to interface with the AbioWire.

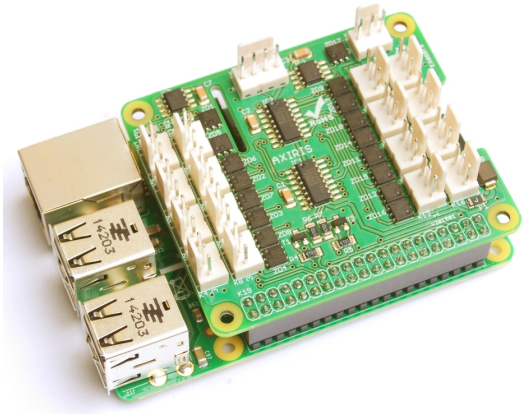
You can deploy the AbioWire with an AxiCat as a combo. For example, let's plug the combo into a Windows system and let the server take control:

```
adapter "abiowire" add serial "\\.\COM3" axicat twi abiowire
```



In the free version of the server, enumeration is limited to the first channel of the first controller. The connector of this channel is labeled "K1" on the AbioWire. The orange marker in the image to the right shows the physical location of the channel's connector.

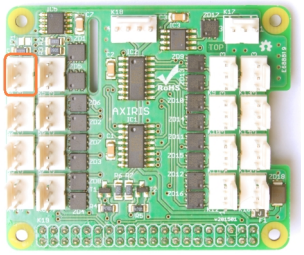
AbioWire+ 1-Wire Adapter



The AbioWire+ is designed for use with the Raspberry Pi A+, B+ and 2 single board computers.

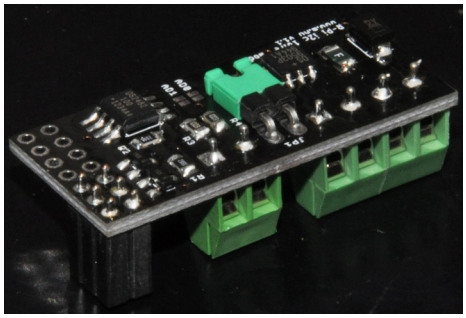
The adapter can be used in conjunction with the AxiCat adapter meaning you can connect the AbioWire+ to a multitude of computer systems.

The AbioWire+ is software-compatible with the AbioWire. To add the adapter to the server's topology, you can use the same **Adapter Add** commands as with the AbioWire.



In the free version of the server, enumeration is limited to the first channel of the first controller. The connector of this channel is labeled "K1" on the AbioWire+. The orange marker in the image to the left shows the physical location of the channel's connector.

m.nu 1-Wire Adapter



This adapter incorporates a DS2482-100 chip as the 1-Wire master. It's designed to fit on a Raspberry Pi single board computer.

Up to four adapters can be stacked on a single Raspberry Pi. Solder pads AD0 and AD1 on the circuit board provide a way to configure the I2C slave address of the DS2482-100 chip. The address range is 18h..1Bh.

Example:

```
adapter "mnu" add bscdetect mnu0
```

You can choose from the following labels to specify the I2C slave address in the **Adapter Add** command:

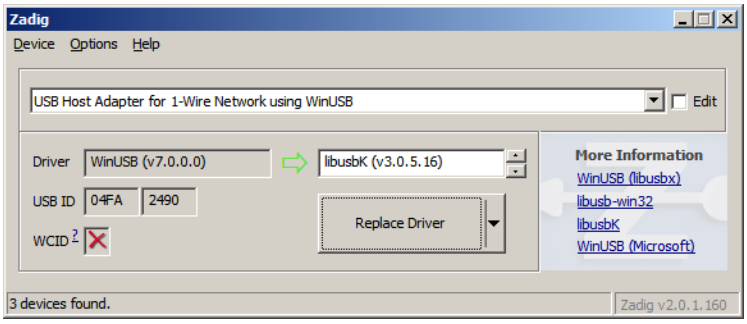
Label	mnu	mnu0	mnu1	mnu2	mnu3
I2C Slave Address	1Bh	1Bh	1Ah	19h	18h

DS2490 USB to 1-Wire Controller

The DS2490 chip implements a USB device function and a 1-Wire master. It can be found in a number of adapters like the DS9490B and DS9490R.

The server relies on LibUSB for driving the DS2490 chip. You don't have to issue command **Adapter Add**, the server automatically adds a device node when a DS2490-based 1-Wire adapter is plugged into a USB port. Just make sure the LibUSB subsystem is enabled.

When you're using a DS2490-based 1-Wire adapter in Windows, use the Zadig tool to update the driver of the 1-Wire adapter. The USB vendor ID and product ID of the DS2490 are 04FAh/2490h. We recommend installing the libusbK driver.



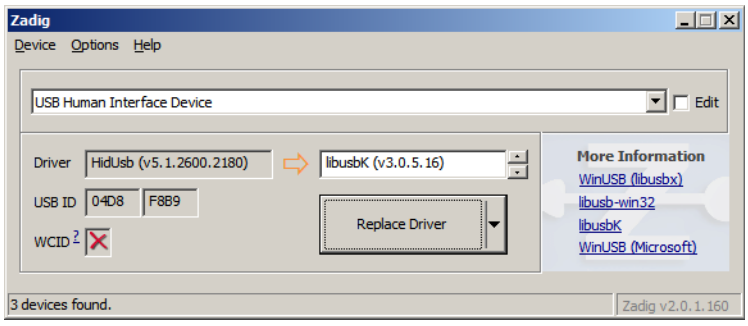
DS9490R/DS9490B 1-Wire Adapter

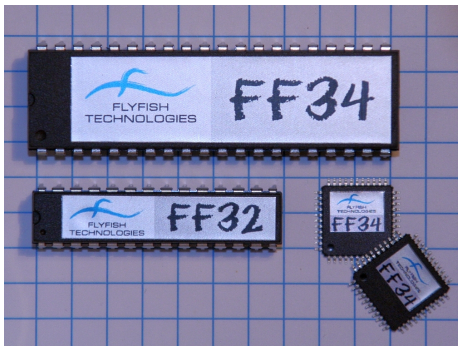
These adapters are based on the DS2490 USB to 1-Wire controller chip.

The DS9490R exposes an RJ12 connector. This adapter usually comes with a 1-Wire serial number chip (family code 81h).

The DS9490R is designed to hold an iButton.

Flyfish FF32/FF34 1-Wire Master





The Flyfish FF32 and FF34 are versatile USB-based interface adapters that offer a variety

of features like digital I/O pins, PWM, I2C master, and a 1-Wire master.

The FF32 and FF34 have the ability to use each I/O pin as the 1-Wire bus. There are two defined ways to assign the I/O pin:

- The chosen I/O pin is stored in non-volatile memory in the device, as a setting. As soon as 1-Wire activity occurs, the device uses the I/O pin as the 1-Wire bus.
- The server explicitly assigns an I/O pin as the 1-Wire bus.

The server supports both.

Each FF32/FF34 stores a chip address (called "USB address" in the FF32/FF34 documentation) in non-volatile memory. The server reads this chip address to look up information for assigning an I/O pin as the 1-Wire bus. If the information is found, the server sends the command to assign the I/O pin, else the server assumes the FF32/FF34 will use the I/O pin stored in non-volatile memory. You can change the chip address using the FF3x Demo program from Flyfish.

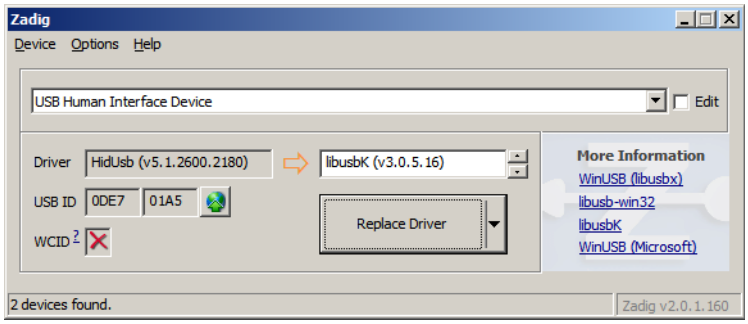
The server defines client commands for controlling the assignment of the I/O pin. Examples:

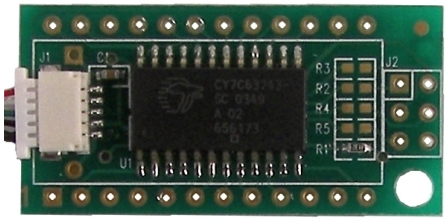
`ff32 ow 10 "B1"`	The server instructs the FF32 with chip address 10 to use pin B1.
`ff32 ow 10 ""`	The server assumes the FF32 with chip address 10 assigns a pin as 1-Wire bus.
`ff34 ow 28 "C5"`	The server instructs the FF34 with chip address 28 to use pin C5.
`ff34 ow 28 ""`	The server assumes the FF34 with chip address 28 assigns a pin as 1-Wire bus.

The server relies on LibUSB for driving the FF32 and FF34. You don't have to issue command **Adapter Add**, the server automatically adds a device node when an FF32 or FF34 is plugged into a USB port. Just make sure the LibUSB subsystem is enabled.

When you're using a Flyfish FF32 or FF34 in Windows, use the Zadig tool to update the driver of the interface adapter. The USB vendor ID and product ID of the FF32/FF34 are 04D8h/F8B9h. We recommend installing the libusbK driver.

USBMicro U401/U421/U451 1-Wire Master





The USBMicro U401/U421/U451 USB-based adapters offer a number of interfaces including a 1-Wire master. Two 8-bit port, A and B, expose all I/O pins. The adapter is capable of using all 16 pins as the 1-Wire DQ line, be it one pin at a time, meaning each I/O pin can act as a 1-Wire channel.

The server defines client commands for controlling the assignment of I/O pins as 1-Wire channels. You can assign I/O pins for a specific adapter based on its USB serial number, and as a default setting for all adapters. Examples:

`u401 ow default "B1"`	Assign pin B1 to U401 adapters by default.
`u421 ow "150211144718" "A4 A5"`	Assign pin A4 and A5 to the U421 adapter with serial number "**150211144718**".
`u421 ow default "A4 B7 A5 B2"`	Assign pins A4, A5, B2 and B7 to U421 adapters by default.
`u451 ow default "A0 A1"`	Assign pins A0 and A1 to U451 adapters by default.

The number of channels is dynamic, based on the pin assignment. The maximum number of channels is sixteen.

Notes that the default assignment for all U401/U421/U451 adapters is pin A0, resulting in a single 1-Wire channel.

The server relies on LibUSB for driving the U401/U421/U451. You don't have to issue command **Adapter Add**, the server automatically adds a device node when a U401/U421/U451 is plugged into a USB port. Just make sure the LibUSB subsystem is enabled.

When you're using a U401/U421/U451 in Windows, use the Zadig tool to update the driver of the interface adapter. The USB vendor ID and product ID are 0DE7h/0191h for the U401, 0DE7h/01A5h for the U421, and 0DE7h/01C5h for the U451. We recommend installing the libusbK driver.

OW-SERVER-ENET-2

The OW-SERVER-ENET-2 device by EDS is a standalone Ethernet server with built-in 1-Wire master. The device exposes three channels with RJ12 jacks for connecting 1-Wire networks.

The device can operate autonomously as a 1-Wire system that supports a variety of 1-Wire slaves.

When the server connects to the device, it bypasses all high-level functionality and directly controls the 1-Wire bus using the low-level interface.

You've to configure OW-SERVER-ENET-2 before the server can connect with the device. The configuration procedure is explained in the documentation of the OW-SERVER-ENET-2. The following settings are of importance to the server:

- IP address of the OW-SERVER-ENET-2.
- Port of the low-level interface.
- The low-level interface must be enabled.

Example:

```
adapter "enet" add enet "192.168.1.101" 8080
```

If your OW-SERVER-ENET-2 has a hostname assigned to it, you can specify the hostname instead of the IP address:

```
adapter "enet" add enet "owserverenet2.mydomain.net" 8080
```

In the free version of the server, enumeration is limited to the first channel of the first controller. This channel is exposed as the RJ12 connector labeled "1" on the OW-SERVER-ENET-2.

TMEX Adapters

If the server's TMEX subsystem is enabled, you can add adapters that are driven by the TMEX software.

TMEX adapters are identified with a port type (0..15) and a port number (0..15). The port type denotes the interface type of the 1-Wire adapter (USB for example), the port number distinguishes between adapters with the same interface.

Port Type	Interface	Adapters
1	Serial	DS9097, DS9097E, DS1413, other compatible passive serial adapters
2	Parallel	DS1410E
5	Serial	DS2480 chip, DS9097U variants
6	USB	DS2490 chip, DS9490R, DS9490B

The mapping of port number to actual adapter depends on the interface type. Read the TMEX documentation for more information.

Example:

```
adapter add tmex 1 4
```

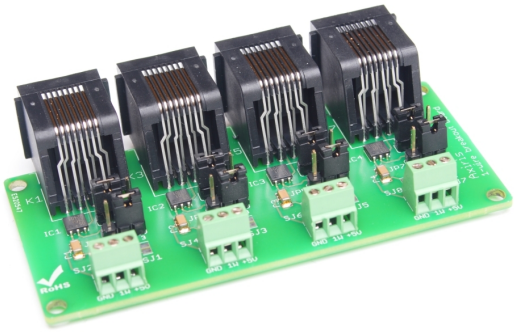
This command adds a passive serial adapter connected to serial port \\.\COM4.

14 1-Wire Slaves

DS2401/2411/1990A Silicon Serial Number

The sole purpose of this 1-Wire slave chip is to associate a unique serial number (ROM code) with a 1-Wire network.

The server's detection procedure uses this family of 1-Wire slave chips as its primary target in the search for 1-Wire networks. When an unallocated channel contains one or more serial number chips, the detection procedure will first search for these chips on the physical 1-Wire bus.



The Axiris 1-Wire Breakout Board (pictured to the right) incorporates four serial number chips, one per breakout path, that accommodate for the detection of up to four 1-Wire networks.

Family code	**01h**
Sensor ID	None
PIO pins	None

DS18S20/DS1920 Thermometer

Family code	**10h**
Sensor ID	**ds18s20**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "10-000802A49A17" sensed ds18s20 "2014-11-12 02:44:17 0026 +019.0 C
parasite"
```

Sensor data information:

Sensor Data String	
`0026 +019.0 C parasite`	
1 **2** **3**	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Power mode: **parasite** or **external**.
Short responses	
temp	Temperature.
pwrmode	Power mode.

DS2406/DS2407 1Kb EPROM Dual Switch

Family code	**12h**
Sensor ID	**ds2406**
PIO pins	**1 or 2**

Example **Device Sensed** responses:

```
dev "12-000000A22310" sensed ds2406 "2014-11-12 02:49:35 0 - 1 - 1 - 1
external"
dev "12-000000A2230F" sensed ds2406 "2014-11-12 02:49:40 0 0 1 1 1 1 2
parasite"
```

Sensor data information:

Sensor Data String	
0 0 1 1 1 1 2 parasite **1 2 3 4 5 6 7 8**	
1	PIO-A sensed state, 0 or 1.
2	PIO-B sensed state, – or 0 or 1.
3	PIO-A output state, 0 or 1.
4	PIO-B output state, – or 0 or 1.
5	PIO-A activity state, 0 or 1.
6	PIO-B activity state, – or 0 or 1.
7	Number of PIO pins, 1 or 2. If this value is 1, fields **2**, **4** and **6** contain a – sign.
8	Power mode: **parasite** or **external**.
Short responses	
piosensed	PIO pins sensed state.
pioset	PIO pins output state.
pioact	PIO pins activity state.
piocnt	Number of PIO pins, 1 or 2.
pwrmode	Power mode.

DS28E04-100 4Kb EEPROM with PIO

Family code	**1Ch**
Sensor ID	**ds28e04**
PIO pins	**2**

Example **Device Sensed** response:

```
dev "1C-00000B8DD77F" sensed ds28e04 "2014-11-12 02:33:45 0 0 1 1 1 1
external"
```

Sensor data information:

Sensor Data String	
0 0 1 1 1 1 external **1 2 3 4 5 6 7**	
1	PIO pin P0 sensed state, 0 or 1.
2	PIO pin P1 sensed state, 0 or 1.
3	PIO pin P0 output state, 0 or 1.
4	PIO pin P1 output state, 0 or 1.
5	PIO pin P0 activity state, 0 or 1.
6	PIO pin P1 activity state, 0 or 1.
7	Power mode: **parasite** or **external**.
Short responses	
piosensed	PIO pins sensed state.
pioset	PIO pins output state.
pioact	PIO pins activity state.
pwrmode	Power mode.

DS2423 4Kb SRAM with counters

Family code	**1Dh**
Sensor ID	**ds2423**
PIO pins	**None**

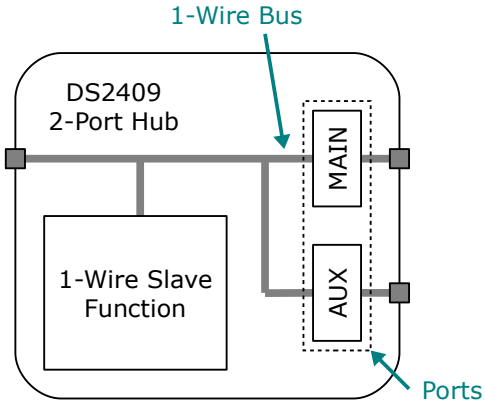
Example **Device Sensed** response:

```
dev "1D-0000000DAC01" sensed ds2423 "2014-11-12 02:52:28 0000000072
0000000032"
```

Sensor data information:

Sensor Data String	
0000000072 0000000032	
1 **2**	
1	Counter A, decimal digits.
2	Counter B, decimal digits.
Short responses	
cntra	Counter A.
cntrb	Counter B.

DS2409 MicroLan Coupler



The DS2409 chip adds two sub-branches to a 1-Wire network. The first port is called MAIN, the second one is called AUX.

Family code	**1Fh**
Sensor ID	**None**
PIO pins	**None**

When you add a DS2409 slave explicitly, you've to specify the type of hub in the **Device Add** client command:

```
dev "EF-15207BA3" add ds2409
```

DS2450 Quad A/D Converter

Family code	**20h**
Sensor ID	**ds2450**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "20-00000014C3CF" sensed ds2450 "2014-11-12 02:41:13 0305 0.0601 V
0333 0.0637 V 0308 0.0603 V 0308 0.0603 V parasite"
```

Sensor data information:

Sensor Data String	
`0305 0.0601 V 0333 0.0637 V 0308 0.0603 V 0308 0.0603 V parasite`	
1 2 3 4 5 6 7 8 9	
1	AIN-A pin conversion result register, hexadecimal digits.
2	Measured voltage on AIN-A pin, decimal representation of **1**.
3	AIN-B pin conversion result register, hexadecimal digits.
4	Measured voltage on AIN-B pin, decimal representation of **3**.
5	AIN-C pin conversion result register, hexadecimal digits.
6	Measured voltage on AIN-C pin, decimal representation of **5**.
7	AIN-D pin conversion result register, hexadecimal digits.
8	Measured voltage on AIN-D pin, decimal representation of **7**.
9	Power mode: **parasite** or **external**.
Short responses	
aina	Measured voltage on AIN-A pin.
ainb	Measured voltage on AIN-B pin.
ainc	Measured voltage on AIN-C pin.
aind	Measured voltage on AIN-D pin.
pwrmode	Power mode.

The DS2450 can be wired for parasite power mode or external power mode in an electronic circuit. However, unlike other chips like the DS18B20, the DS2450 can't detect the effective power mode. Instead, the host must tell the chip which power mode is in effect. The server provides client command **Device Attribute** for this purpose. For example:

```
dev "20-14C3CF" attr pwrmode=external
```
Set chip in external power mode.

```
dev "20-14C3CF" attr pwrmode=parasite
```
Set chip in parasite power mode.

The default is parasite power mode.

DS1822 Econo Digital Thermometer

Family code	**22h**
Sensor ID	**ds1822**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "22-0000003201DA" sensed ds1822 "2014-11-12 02:45:12 013D +019.8 C
parasite"
```

Sensor data information:

Sensor Data String	
`013D +019.8 C parasite`	
1 **2** **3**	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Power mode: **parasite** or **external**.
Short responses	
temp	Temperature.
pwrmode	Power mode.

DS2438 Smart Battery Monitor

Family code	**26h**
Sensor ID	**ds2438**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "26-00000141BFCE" sensed ds2438 "2014-11-12 02:35:30 1440 +020.3 C
0000 00.00 V 01F3 04.99 V FFF9 -0007"
```

Sensor data information:

Sensor Data String	
`1440 +020.3 C 0000 00.00 V 01F3 04.99 V FFF9 -0007`	
1 2 3 4 5 6 7 8	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Voltage register, measured on pin V_{DD}, hexadecimal digits.
4	Voltage measured on pin V_{DD}, decimal representation of **3**.
5	Voltage register, measured on pin V_{AD}, hexadecimal digits.
6	Voltage measured on pin V_{AD}, decimal representation of **5**.
7	Current register, hexadecimal digits.
8	Current, decimal representation of **7**.
Short responses	
temp	Temperature.
vdd	Voltage measured on pin V_{DD}.
vad	Voltage measured on pin $V_{AD.}$
cur	Current.

DS18B20 Thermometer

Family code	**28h**
Sensor ID	**ds18b20**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "28-0000040CBBB2" sensed ds18b20 "2014-11-12 02:44:47 0132 +019.1 C
parasite"
```

Sensor data information:

Sensor Data String	
`0132` `+019.1 C` `parasite`	
1 **2** **3**	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Power mode: **parasite** or **external**.
Short responses	
temp	Temperature.
pwrmode	Power mode.

DS2408 8-Channel Addressable Switch

Family code	**29h**
Sensor ID	**ds2408**
PIO pins	**8**

Example **Device Sensed** response:

```
dev "29-00000011BD2A" sensed ds2408 "2014-11-12 02:37:51 00000000 11111111
00000000 reset  external"
```

Sensor data information:

Sensor Data String	
`00000000 11111111 00000000 reset external`	
1 2 3 4 5	
1	PIO pins sensed state, binary digits.
2	PIO pins output state, binary digits.
3	PIO pins activity state, binary digits.
4	RSTZ pin configuration: ▪ **reset**: the pin is configured as $\overline{\text{RST}}$ input. ▪ **strobe**: the pin is configured as $\overline{\text{STRB}}$ output.
5	Power mode: **parasite** or **external**.
Short responses	
piosensed	PIO pins sensed state.
pioset	PIO pins output state.
pioact	PIO pins activity state.
rstz	RSTZ pin configuration.
pwrmode	Power mode.

DS2760/2761/2762 Li+ Battery Monitor

Family code	**30h**
Sensor ID	**ds2760**
PIO pins	**1**

Example **Device Sensed** response:

```
dev "30-000012B5735B" sensed ds2760 "2014-11-12 02:44:28 1320 +019.1 C
7730 +4.65 V 8000 -4096 3CF8 +15608 1"
```

Sensor data information:

Sensor Data String	
`1320 +019.1 C 7730 +4.65 V 8000 -4096 3CF8 +15608 1`	
1 2 3 4 5 6 7 8 9	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Voltage register, hexadecimal digits.
4	Voltage measured on pin V_{IN}, decimal representation of **3**.
5	Current register, hexadecimal digits.
6	Current, decimal representation of **5**.
7	Current accumulator register, hexadecimal digits.
8	Current accumulator, decimal representation of **7**.
9	PIO pin sensed state, 0 or 1.
Short responses	
temp	Temperature.
vin	Voltage measured on pin V_{IN}.
cur	Current.
curacc	Current accumulator.
piosensed	PIO pin sensed state.

DS2780 Standalone Fuel Gauge

Family code	**32h**
Sensor ID	**ds2780**
PIO pins	**1**

Example **Device Sensed** response:

```
dev "32-0000006BDBC0" sensed ds2780 "2014-11-12 02:33:54 14C0 +020.8 C
54A0 +3.30 V 0063 +00012 0037 DAE0 003607982 0"
```

Sensor data information:

Sensor Data String	
`14C0 +020.8 C 54A0 +3.30 V 0063 +00012 0037 DAE0 003607982 0`	
1 2 3 4 5 6 7 8 9 10	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Voltage register, hexadecimal digits.
4	Voltage measured on pin V_{IN}, decimal representation of **3**.
5	Current register, hexadecimal digits.
6	Current, decimal representation of **5**.
7	Current accumulator register, most significant 16 bits, hexadecimal digits.
8	Current accumulator register, least significant 12 bits, 4 undefined bits, hexadecimal digits.
9	Current accumulator, decimal representation of **8** and **9**, 28-bit value.
10	PIO pin sensed state, 0 or 1.
Short responses	
temp	Temperature.
vin	Voltage measured on pin V_{IN}.
cur	Current.
curacc	Current accumulator.
piosensed	PIO pin sensed state.

DS2755/2756 High-Precision Battery Fuel Gauge

Family code	**35h**
Sensor ID	**ds2755**
PIO pins	**1**

Example **Device Sensed** response:

```
dev "35-00000043F47C" sensed ds2755 "2014-11-12 02:34:24 139A +019.5 C
54BE +3.30 V 0011 +0002 00F1 +00241 1"
```

Sensor data information:

Sensor Data String	
`139A +019.5 C 54BE +3.30 V 0011 +0002 00F1 +00241 1`	
1 2 3 4 5 6 7 8 9	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Voltage register, hexadecimal digits.
4	Voltage measured on pin V_{IN}, decimal representation of **3**.
5	Current register, hexadecimal digits.
6	Current, decimal representation of **5**.
7	Current accumulator register, hexadecimal digits.
8	Current accumulator, decimal representation of **7**.
9	PIO pin sensed state, 0 or 1.
Short responses	
temp	Temperature.
vin	Voltage measured on pin V_{IN}.
cur	Current.
curacc	Current accumulator.
piosensed	PIO pin sensed state.

DS2740 Coulomb Counter

Family code	**36h**
Sensor ID	**ds2740**
PIO pins	**1**

Example **Device Sensed** response:

```
dev "36-000003612CDB" sensed ds2740 "2014-11-12 02:34:08 0115 +00277 009C
+00156 0"
```

Sensor data information:

Sensor Data String	
0115 +00277 009C +00156 0	
1 **2** **3** **4** **5**	
1	Current register, hexadecimal digits.
2	Current, decimal representation of **1**.
3	Current accumulator register, hexadecimal digits.
4	Current accumulator, decimal representation of **3**.
5	PIO pin sensed state, 0 or 1.
Short responses	
cur	Current.
curacc	Current accumulator.
piosensed	PIO pin sensed state.

DS2413 Dual-Channel Addressable Switch

Family code	**3Ah**
Sensor ID	**ds2413**
PIO pins	**2**

Example **Device Sensed** responses:

```
dev "3A-000000052F6A" sensed ds2413 "2014-11-12 02:45:28 0 0 1 1"
```

Sensor data information:

Sensor Data String	
0 0 1 1 **1 2 3 4**	
1	PIO-A sensed state, 0 or 1.
2	PIO-B sensed state, 0 or 1.
3	PIO-A output state, 0 or 1.
4	PIO-B output state, 0 or 1.
Short responses	
piosensed	PIO pins sensed state.
pioset	PIO pins output state.

DS1825 Thermometer

Family code	**3Bh**
Sensor ID	**ds1825**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "3B-00000019CCEF" sensed ds1825 "2014-11-12 02:35:22 B 0135 +019.3 C
external"
```

Sensor data information:

Sensor Data String	
`B 0135 +019.3 C external`	
1 2 3 4	
1	Address pins AD[3..0], hexadecimal digit.
2	Temperature register, hexadecimal digits.
3	Temperature, decimal representation of **2** in client temperature scale.
4	Power mode: **parasite** or **external**.
Short responses	
adpins	Address pins.
temp	Temperature.
pwrmode	Power mode.

When you add this chip explicitly, you've to specify the sensor identifier in the **Device Add** client command:

```
dev "3B-00000019CCEF" add ds1825
```

MAX31826 Thermometer

Family code	**3Bh**
Sensor ID	**max31826**
PIO pins	**None**

Example **Device Sensed** response:

```
dev "3B-0000001529B5" sensed max31826 "2014-11-12 02:34:55 0 0139 +019.6 C
external"
```

Sensor data information:

Sensor Data String	
`0 0139 +019.6 C external` **1 2 3 4**	
1	Address pins AD[3..0], hexadecimal digit.
2	Temperature register, hexadecimal digits.
3	Temperature, decimal representation of **2** in client temperature scale.
4	Power mode: **parasite** or **external**.
Short responses	
adpins	Address pins.
temp	Temperature.
pwrmode	Power mode.

When you add this chip explicitly, you've to specify the sensor identifier in the **Device Add** client command:

```
dev "3B-0000001529B5" add max31826
```

MAX31850 Thermocouple

Family code	**3Bh**
Sensor ID	**max31850**
PIO pins	**None**

Example **Device Sensed** responses:

```
dev "3B-000000183368" sensed max31850 "2014-11-12 02:45:48 0 0130 0
+0019.00 C 1380 0 0 0 +019.5 C external"
dev "3B-000000183368" sensed max31850 "2014-11-12 22:18:56 0 7FFD 1
UNCONN     13A1 1 0 0 +019.6 C external"
```

Sensor data information:

Sensor Data String	
`0 0130 0 +0019.00 C 1380 0 0 0 +019.5 C external`	
1 2 3 4 5 6 7 8 9 10	
1	Address pins AD[3..0], hexadecimal digit.
2	Thermocouple hot temperature register, hexadecimal digits.
3	Fault detected y/n, 0 or 1.
4	Depends on whether the thermocouple is connected: ▪ Thermocouple hot temperature, decimal representation of **2** in client temperature scale. ▪ **UNCONN** indicates the thermocouple isn't connected.
5	Internal cold junction temperature register, hexadecimal digits.
6	Open circuit detected y/n, 0 or 1.
7	Short to GND pin detected y/n, 0 or 1.
8	Short to VCC pin detected y/n, 0 or 1.
9	Internal cold junction temperature, decimal representation of **5** in client temperature scale.
10	Power mode: **parasite** or **external**.
Short responses	
adpins	Address pins.
tcfault	Fault detected y/n.
tcunconn	Thermocouple is unconnected y/n.
tchot	Thermocouple hot temperature. Value is valid only if **tcunconn** returns 0.
tcopen	Open circuit detected y/n.
tcgnd	Short to GND pin detected y/n.
tcvcc	Short to VCC pin detected y/n.
tccold	Internal cold junction temperature.
pwrmode	Power mode.

When you add this chip explicitly, you've to specify the sensor identifier in the **Device Add** client command:

```
dev "3B-000000183368" add max31850
```

DS2775/2776/2781 Li+ Fuel Gauge

Family code	**3Dh**
Sensor ID	**ds2781**
PIO pins	**1**

Example **Device Sensed** response:

```
dev "3D-00000081D206" sensed ds2781 "2014-11-12 02:34:46 1420 +020.1 C
2A40 +3.30 V 0020 +00032 0024 0000 000000036 0"
```

Sensor data information:

Sensor Data String	
`1420 +020.1 C 2A40 +3.30 V 0020 +00032 0024 0000 000000036 0`	
1 2 3 4 5 6 7 8 9 10	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	Voltage register, hexadecimal digits.
4	Voltage measured on pin V_{IN}, decimal representation of **3**.
5	Current register, hexadecimal digits.
6	Current, decimal representation of **5**.
7	Current accumulator register, most significant 16 bits, hexadecimal digits.
8	Current accumulator register, least significant 12 bits, 4 undefined bits, hexadecimal digits.
9	Current accumulator, decimal representation of **8** and **9**, 28-bit value.
10	PIO pin sensed state, 0 or 1.
Short responses	
temp	Temperature.
vin	Voltage measured on pin V_{IN}.
cur	Current.
curacc	Current accumulator.
piosensed	PIO pin sensed state.

DS28EA00 Thermometer

Family code	**42h**
Sensor ID	**ds28ea00**
PIO pins	**2**

Example **Device Sensed** responses:

```
dev "42-00000038D0BE" sensed ds28ea00 "2014-11-12 02:44:58 0133 +019.2 C 1
1 1 1 external"
```

Sensor data information:

Sensor Data String	
`0133 +019.2 C 1 1 1 1 external`	
1 2 3 4 5 6 7	
1	Temperature register, hexadecimal digits.
2	Temperature, decimal representation of **1** in client temperature scale.
3	PIO-A sensed state, 0 or 1.
4	PIO-B sensed state, 0 or 1.
5	PIO-A output state, 0 or 1.
6	PIO-B output state, 0 or 1.
7	Power mode: **parasite** or **external**.
Short responses	
temp	Temperature.
piosensed	PIO pins sensed state.
pioset	PIO pins output state.
pwrmode	Power mode.

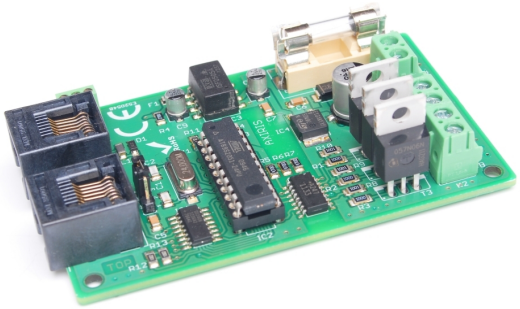
DS1420 Serial ID Button

This chip is often found in the DS9490R USB to 1-Wire adapter.

The server's detection procedure uses this family of 1-Wire slave chips as a target in the search for 1-Wire networks. When an unallocated channel contains one or more members of this family, the detection procedure will search for these chips on the physical 1-Wire bus.

Family code	**81h**
Sensor ID	**None**
PIO pins	**None**

Axiris 1-Wire RGB Controller



The 1-Wire RGB Controller by Axiris offers three individually controllable PWM output channels able to directly drive a 12V LED RGB strip. The generated PWM signals have a frequency of approx. 1000 Hz and an adjustable duty cycle from 0 % to 99.61 %. PWM signals are shifted 120 degrees for optimal power distribution.

The device embeds a DS2408 as the 1-Wire slave function.

Before the RGB channels can be programmed, the RSTZ pin of the DS2408 must be configured as STRB output. For example:

```
dev "29-11CEE1" pio rstz strobe
```

Use client command **Device RGB Controller** to control the RGB channels. Examples:

```
dev "29-11CEE1" rgbctrl red=100
```

Turn on the red channel at 100/256 = 39 %.

```
dev "29-11CEE1" rgbctrl red=64 green=128 blue=192
```

Turn on the RGB channels: red at 25 %, green at 50 %, blue at 75 %.

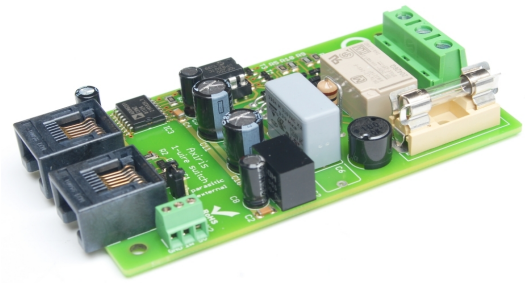
You can set channel values in advance and turn on and off channels later. For example:

```
dev "29-11CEE1" rgbctrl red=off,200 green=off,200 blue=off,200
dev "29-11CEE1" rgbctrl red=on
dev "29-11CEE1" rgbctrl green=on blue=on red=off
```

The following command sets all channels to zero and turns them off.

```
dev "29-11CEE1" rgbctrl clear
```

Axiris 1-Wire Mains Switch



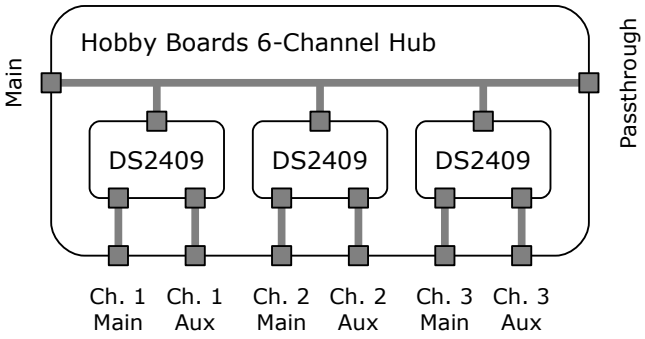
The 1-Wire Mains Switch by Axiris is designed to switch a 120 V or 230 V load remotely over a 1-Wire bus. Applications include switching on and off lights, home automation, and industrial automation.

The device embeds a DS2406 as the 1-Wire slave function.

You can use client command **Device Switch** to control the device. Examples:

`dev "12-974696" switch on`	Turn on the switch.
`dev "12-974696" switch off`	Turn off the switch.

Hobby Boards 6-Channel Hub



The Hobby Boards 6-Channel Hub incorporates three DS2409 chips. The hub adds six sub-branches to a 1-Wire network.

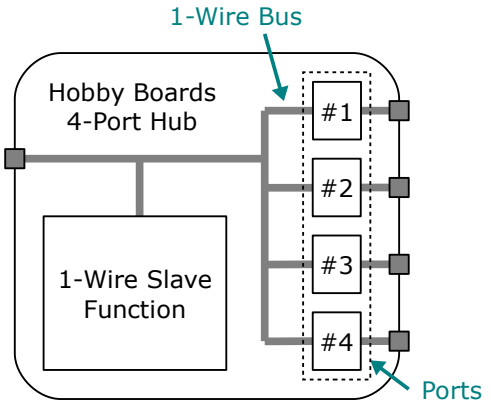
When you add the hub explicitly, you've to specify the type of hub in the **Device Add** client command:

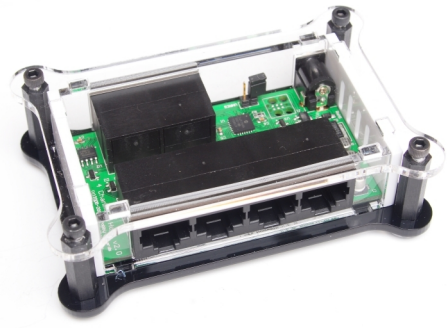
```
dev "1F-56B3E" add ds2409
dev "1F-56B31" add ds2409
dev "1F-56BA7" add ds2409
```

Example topology dump:

```
dump
01: USB to 1-Wire "usb-4-2"
  01: DS2490
    01: 81-000000324BBD-31  DS1420 serial ID button
    01: 1F-000000056B3E-A7  DS2409 MicroLan coupler
    01: 1F-000000056B31-83  DS2409 MicroLan coupler
    01: 1F-000000056BA7-80  DS2409 MicroLan coupler
```

Hobby Boards 4-Channel Hub





The Hobby Boards 4-Channel Hub adds four sub-branches to a 1-Wire network. The ports are numbered 1 to 4.

Family code	**EFh** ▪ **Hobby Boards type 05h**
Sensor ID	**None**
PIO pins	**None**

When you add the hub explicitly, you've to specify the type of hub in the **Device Add** client command:

```
dev "1F-56BA7" add hbh4
```

15 Server Versions

Feature	Free Version	Full Version
Enumeration procedure	LIMITED[1]	YES
Detection procedure	YES	YES
Limiting the number of connections	NO	YES
User authentication	NO	YES
List of allowed client IP addresses	NO	YES
Loading and saving topology files	NO	YES
Moving device groups	NO	YES
Clearing device groups	NO	YES
Supported 1-Wire adapters	ALL	ALL
Supported 1-Wire slaves	ALL	ALL
Program instances	ONE	UNLIMITED
Number of connected adapters	ONE	UNLIMITED

[1] Enumeration is limited to the first channel of the first controller and stops after finding three 1-Wire slaves.

16 Software Revision History

Changes apply to all program versions unless specified otherwise.

Version	Description
1.0.0	▪ Initial release.
1.1.0	▪ Added USBMicro U401/U421/U451 1-Wire master. ▪ Added DS9097 passive serial adapter. ▪ Added W1 subsystem. ▪ Added TMEX connector. ▪ Added support for owfs-style ROM codes in client commands and responses. ▪ Added command line argument -romcode. ▪ Fixed precedence of command line option -port. It now overrules the port setting in the configuration file as stated in the user manual. ▪ Fixed a crash that may occur when a 1-Wire controller is disabled while multiple client commands are trying to access the controller.

17 Legal Information

Disclaimer

Axiris products are not designed, authorized or warranted to be suitable for use in space, nautical, space, military, medical, life-critical or safety-critical devices or equipment.

Axiris products are not designed, authorized or warranted to be suitable for use in applications where failure or malfunction of an Axiris product can result in personal injury, death, property damage or environmental damage.

Axiris accepts no liability for inclusion or use of Axiris products in such applications and such inclusion or use is at the customer's own risk. Should the customer use Axiris products for such application, the customer shall indemnify and hold Axiris harmless against all claims and damages.

Trademarks

"Maxim Integrated" is a trademark of Maxim Integrated Products, Inc.

"1-Wire" and "iButton" are registered trademarks of Maxim Integrated Products, Inc.

"Raspberry Pi" is a trademark of the Raspberry Pi Foundation.

All product names, brands, and trademarks mentioned in this document are the property of their respective owners.

18 Contact Information

Official website: http://www.axiris.eu/