cbus Documentation

Release 0.2-dev

Michael Farrell

Oct 24, 2021

Contents

Project Page / Source repository: https://github.com/micolous/cbus

libcbus is a set of Python libraries for interacting with Clipsal C-Bus.

Contents:

CHAPTER 1

Introduction

Welcome to `libcbus`!

This is a Python library for interacting with Clipsal C-Bus networks through a PCI (PC Interface) or CNI (C-Bus Network Interface).

This consists of:

- *A C-Bus MQTT bridge (cmqttd)*, which provides a high level API for controlling C-Bus networks with other systems (such as Home Assistant)
- A low-level interface for parsing and producing C-Bus packets, and using a PCI with `asyncio`
- A library for parsing information from C-Bus Toolkit project backup files, and visualising networks with **`graphviz`**
- A "fake PCI" test server for parsing data sent by C-Bus applications.

It is a completely open source implementation (LGPLv3) of the C-Bus PCI/CNI protocol in Python, based on Clipsal's public documentation of the PCI Serial Interface and some reverse engineering.

Unlike a number of other similar projects, it *does not* depend on *C-Gate or libcbm*. This makes the code much more portable between platforms, as well as avoiding the hazards of closed-source software. :)

> **Warning:** Despite using RJ45 connectors and CAT-5 cabling commonly associated with Ethernet networks, C-Bus uses totally different signalling (about 10 kbit/s) and has a 36 volt power feed.
>
> _You cannot patch an ordinary network card into a C-Bus network._
>
> This project *requires* a PCI or CNI to communicate with a C-Bus network.

1.1 What is C-Bus?

C-Bus is a home automation and electrical control system made by Clipsal. It's also known as Square D in the United States, and sold under other brands worldwide by Schnider Electric.

It uses low voltage (36 volts) wiring for light switches (panels) and other sensors, and centrally-fed dimmer and relay controls for devices (such as lights).

The C-Bus PCI and CNI can interface with a C-Bus network via Serial[1] and TCP/IPv4 respectively. These use a common interface described in the Serial Interface Guide, and other public C-Bus documentation.

1.2 Clipsal's other interfaces

In addition to protocol documentation, Clipsal also provide two systems for interacting with C-Bus, `libcbm` and C-Gate. Clipsal's own software (like Toolkit) and hardware (like Wiser) use this to interact with C-Bus networks over serial and IPv4.

1.2.1 `libcbm`

`libcbm` supports to C-Bus protocol completely, including conforming to the various "protocol certification levels".

It is written in C, and distributed as a static library for `x86_32` Linux and Windows systems. Clipsal has released its source code under the Boost license, which also includes Delphi bindings and support for ARMv3/4/4T, Hitachi/Renesas H8, PowerPC 405 (on Linux) and TI MSP430 processors.

This hasn't been updated since 2009, and doesn't support `x86_64` or comparatively-modern ARM CPUs (such as that used in the Raspberry Pi).

1.2.2 C-Gate

C-Gate is a closed source, C-Bus abstraction service written in Java.

It appears to support a subset of the C-Bus protocol, and comparing its interactions with a PCI with the Serial Interface Guide seems to suggest it is using a bunch of commands that are officially deprecated.

It depends on the (closed source) SerialIO library for serial communication, which requires a JNI (Java Native Interface) library that is only available on `x86_32` Windows and old versions of Linux.

[1] The PCI is also available in a USB variant, which uses an in-built `cp210x` USB to Serial converter. It is otherwise functionally identical to the Serial version.

CHAPTER 2

Installing libcbus

Note: This section is incomplete.

2.1 All components (system install)

You need Python 3.7 or later installed. You can build the software and its dependencies with:

```
$ pip3 install -r requirements.txt
$ python3 setup.py install
```

This will install everything, including **cmqttd**.

2.2 C-Bus MQTT bridge only (Docker image)

See *Running in Docker*.

CHAPTER 3

cmqttd

cmqttd allows you to expose a C-Bus network to an MQTT broker. This daemon replaces **cdbusd** (which required D-Bus) as the abstraction mechanism for all other components.

It uses Home Assistant style MQTT-JSON Light components, and supports MQTT discovery. It should also work with other software that supports MQTT.

It *can also be run inside a Docker container*.

This replaces **sage** (our custom web interface which replaced *Wiser*).

cmqttd with Home Assistant has many advantages over *Wiser*:

- No dependency on Flash Player or a mobile app
- No requirement for an Ethernet-based PCI (serial or USB are sufficient)
- Touch-friendly UI based on Material components
- Integrates with other Home Assistant supported devices
- No *hard coded back-doors* or outdated software from 2006

See also: *Instructions for Wiser users*.

Note: Only the default lighting application is supported by **cmqttd**. Patches welcome!

3.1 Running

cmqttd requires a MQTT Broker (server) to act as a message bus.

Note: For these examples, we'll assume your MQTT Broker:

- is accessible via `192.0.2.1` on the default port (1883).

- does not use transport security (TLS)
- does not require authentication

This setup is *not* secure; but securing your MQTT Broker is out of the scope of this document.

For more information, see *MQTT options*.

To connect to a serial or USB PCI connected on `/dev/ttyUSB0`, run:

```
$ cmqttd --broker-address 192.0.2.1 --broker-disable-tls --serial /dev/ttyUSB0
```

To connect to a CNI (or PCI over TCP) listening at `192.0.2.2:10001`, run:

```
$ cmqttd --broker-address 192.0.2.1 --broker-disable-tls --tcp 192.0.2.2:10001
```

If you're using Docker, the container also needs a route to the CNI's IP address.

Tip: If you haven't *installed the library*, you can run from a `git clone` of `libcbus` source repository with:

```
$ python3 -m cbus.daemons.cmqttd -b 192.0.2.1 [...]
```

3.1.1 For Wiser users

This software is **not** compatible with Wiser Home Control (Clipsal's web interface for C-Bus). Wiser and **cmqttd** both take full control the CNI, and will interfere with one another.

Additionally, using both on the same C-Bus network (with different PCI/CNIs) may cause issues, as both presume they are the sole source of network services such as time synchronisation.

Wiser Home Control Mk1 (5200PG)

The Wiser Home Control Mk1 has an external CNI which should be usable with **cmqttd**.

1. Switch off and completely disconnect the Wiser.
2. Disconnect the "busbar" between the Wiser and the CNI.
3. Connect the CNI to power and network directly.

   You may need to use Toolkit to configure the CNI with an IP address which can be accessed from the host you're running **cmqttd** on. The default IP address for the CNI is `192.168.2.2`.
4. Continue setting up **cmqttd**.
5. Once you've verified **cmqttd** is working correctly, responsibly dispose of the Wiser 1 at your nearest e-waste facility.

Warning: The Wiser 1 has very outdated and insecure software (from 2006). *You should not use it under any circumstances, or for any purpose.*

Wiser Home Control Mk2 (5200WHC2)

The Wiser Home Control Mk2 has an internal CNI which cannot be used, because the Wiser's software conflicts with **cmqttd**.

You will need to get a real, standalone PCI or CNI.

Tip: The author of this software does not have access to any Wiser hardware anymore, and the Wiser 2's list price of 2000 AUD is far beyond the budget for this project.

Hint hint, Schneider Electric... we should talk :)

3.2 Configuration

cmqttd has many command-line configuration options.

A complete list can be found by running `cmqttd --help`.

3.2.1 C-Bus PCI options

One of these *must* be specified:

--serial DEVICE
: Serial device that the PCI is connected to, eg: `/dev/ttyUSB0`.

  USB PCIs (5500PCU) act as a SiLabs `cp210x` USB-Serial adapter, its serial device must be specified here.

--tcp ADDR:PORT
: IP address and TCP port where the PCI or CNI is located, eg: `192.0.2.1:10001`.

  Both the address and the port are required. CNIs listen on port `10001` by default.

  See also: *Instructions for Wiser users*.

3.2.2 MQTT options

--broker-address ADDR
: Address of the MQTT broker. This option is required.

--broker-port PORT
: Port of the MQTT broker.

  By default, this is 8883 if TLS is enabled, otherwise 1883.

--broker-disable-tls
: Disables all transport security (TLS). This option is insecure!

  By default, transport security is enabled.

--broker-auth FILE
: File containing the username and password to authenticate to the MQTT broker with.

  This is a plain text file with two lines: the username, followed by the password.

  If not specified, password authentication will not be used.

--broker-ca DIRECTORY
Path to a directory of CA certificates to trust, used for validating certificates presented in the TLS handshake.

If not specified, the default (Python) CA store is used instead.

--broker-client-cert PEM

--broker-client-key PEM
Path to a PEM-encoded client (public) certificate and (private) key for TLS authentication.

If not specified, certificate-based client authentication will not be used.

If the file is encrypted, Python will prompt for the password at the command-line.

3.2.3 Labels

--project-file CBZ
Path to a C-Bus Toolkit project backup file (CBZ) to use for labelling group addresses.

This doesn't affect the entity paths or unique IDs published in MQTT.

Only single-network projects using the lighting application are supported. DLT labels are not supported.

For group addresses with unknown names, or if no project file is supplied, generated names like `C-Bus Light 001` will be used instead.

Tip: If you don't have a project file backup from your installer, you can always rename entities from within Home Assistant itself.

This labels are not stored on C-Bus units, so Toolkit cannot download this information from the network.

3.2.4 Time synchronisation

By default, **cmqttd** will periodically provide a time signal to the C-Bus network, and respond to all time requests.

Local time is always used for time synchronisation. You can specify a different timezone with the TZ environment variable.

C-Bus' time implementation has many limitations:

- C-Bus date values and time values are two separate network variables – there is no analog to Python's `datetime.datetime` type. This can trigger race conditions around midnight if the messages are not handled atomically by receivers.

  cmqttd will always send the date and time as a single message, in an attempt to mitigate this issue.

- C-Bus time values have an optional "daylight saving time" flag, with three states: "no daylight saving offset applied", "time advanced by 1 hour for daylight saving", and "unknown".

  Because this is cannot be used to present daylight saving time properly (eg: Lord Howe Island turns their clocks forward 30 minutes for DST), and there are far too many edge cases with time zone handling, **cmqttd** will always report "unknown", in an attempt to make sure C-Bus units do not attempt any time conversions.

- C-Bus does not support leap seconds. You can mitigate this by synchronising your clock using an NTP server with leap second smearing.

To schedule scenes in C-Bus, you should use something like Home Assistant, rather than embedded controllers directly attached to the C-Bus network.

--timesync SECONDS
: Periodically sends an unsolicited time signal to the C-Bus network.

  By default, this is every 300 seconds (5 minutes).

  If set to `0`, **cmqttd** will not send unsolicited time signals to the C-Bus network.

--no-clock
: Disables responding to time requests from the C-Bus network.

3.2.5 Logging

--log-file FILE
: Where to write the log file. If not specified, logs are written to `stdout`.

--verbosity LEVEL
: Verbosity of logging to emit. If not specified, defaults to `INFO`.

  Options: CRITICAL, ERROR, WARNING, INFO, DEBUG

3.3 Using with Home Assistant

cmqttd supports Home Assistant's MQTT discovery protocol.

To use it, just add a MQTT integration using the same MQTT Broker as **cmqttd** with discovery enabled (this is *disabled* by default). See Home Assistant's documentation for more information and example configurations.

Once the integration and **cmqttd** are running, each group addresses (regardless of whether it is in use) will automatically appear in Home Assistant's UI as two components:

- lights: `light.cbus_{{GROUP_ADDRESS}}` (eg: GA 1 = `light.cbus_1`)

  This implements read / write access to lighting controls on the default lighting application. "Lighting Ramp" commands can be sent via the standard `brightness` and `transition` extensions.

  By default, these will have names like `C-Bus Light 001`.

- binary sensors: `binary_sensor.cbus_{{GROUP_ADDRESS}}` (eg: GA 1 = `binary_sensor.cbus_1`).

  This is a binary, read-only interface for all group addresses.

  An example use case is a PIR (occupancy/motion) sensor that has been configured (in C-Bus Toolkit) to actuate two group addresses – one for the light in the room (shared with an ordinary wall switch), and which only reports recent movement.

  cmqttd doesn't assign any class to this component, so this can be used however you like. Any brightness value is ignored.

  By default, these will have names like `C-Bus Light 001 (as binary sensor)`.

All elements can be renamed and customized from within Home Assistant.

3.4 Running in Docker

This repository includes a `Dockerfile`, which uses a minimal Alpine Linux image as a base, and contains the *bare minimum* needed to make **cmqttd** work.

On a system with Docker installed, clone the libcbus git repository and then run:

```
# docker build -t cmqttd .
```

This will download about 120 MiB of dependencies, and result in about 100 MiB image (named `cmqttd`).

The image's startup script (`entrypoint-cmqttd.sh`) uses the following environment variables:

TZ
: The timezone to use when sending a time signal to the C-Bus network.

  This must be a tz database timezone name (eg: `Australia/Adelaide`). The default (and fall-back) timezone is UTC.

SERIAL_PORT
: The serial port that the PCI is connected to. USB PCIs appear as a serial device (`/dev/ttyUSB0`).

  Docker *also* requires the `--device` option so that it is forwarded into the container.

  This is equivalent to *`cmqttd --serial`*. Either this or *`CNI_ADDR`* is required.

CNI_ADDR
: A TCP `host:port` where a CNI is located.

  This is equivalent to *`cmqttd --tcp`*. Either this or *`SERIAL_PORT`* is required.

  See also: *Instructions for Wiser users*.

MQTT_SERVER
: IP address where the MQTT Broker is running.

  This is equivalent to *`cmqttd --broker-address`*. This environment variable is required.

MQTT_PORT
: Port address where the MQTT Broker is running.

  This is equivalent to *`cmqttd --broker-port`*.

MQTT_USE_TLS
: If set to `1` (default), this enables support for TLS.

  If set to `0`, TLS support will be disabled. This is equivalent to *`cmqttd --broker-disable-tls`*.

CBUS_CLOCK
: If set to `1` (default), **cmqttd** will respond to time requests from the C-Bus network.

  If set to `0`, **cmqttd** will ignore time requests from the C-Bus network. This is equivalent to *`cmqttd --no-clock`*.

CBUS_TIMESYNC
: Number of seconds to wait between sending an unsolicited time signal to the C-Bus network.

  If set to `0`, **cmqttd** will not send unsolicited time signals to the C-Bus network.

  By default, this will be sent every 300 seconds (5 minutes).

  This is equivalent to *`cmqttd --timesync`*.

The image is configured to read additional files from `/etc/cmqttd`, if present. Use Docker volume mounts to make the following files available:

/etc/cmqttd/auth Username and password to use to connect to an MQTT broker, separated by a newline character.

  If this file is not present, then **cmqttd** will try to use the MQTT broker without authentication.

  This is equivalent to *`cmqttd --broker-auth`*.

/etc/cmqttd/certificates A directory of CA certificates to trust when connecting with TLS.

If this directory is not present, the default (Python) CA store will be used instead.

This is equivalent to *cmqttd --broker-ca*.

/etc/cmqttd/client.pem, /etc/cmqttd/client.key Client certificate (`pem`) and private key (`key`) to use to connect to the MQTT broker.

This is equivalent to *cmqttd --broker-client-cert* and *cmqttd --broker-client-key*.

/etc/cmqttd/project.cbz C-Bus Toolkit project backup file to use as a source for labelling group addresses.

This is is equivalent to *cmqttd --project-file*.

Note: All file and directory names are case-sensitive, and must be lower case.

3.4.1 Docker usage examples

To use a PCI on `/dev/ttyUSB0`, with an unauthenticated and unencrypted MQTT Broker at `192.0.2.1`, and the time zone set to `Australia/Adelaide`:

```
# docker run --device /dev/ttyUSB0 -e "SERIAL_PORT=/dev/ttyUSB0" \
    -e "MQTT_SERVER=192.0.2.1" -e "MQTT_USE_TLS=0" \
    -e "TZ=Australia/Adelaide" cmqttd
```

To supply MQTT broker authentication details, create an `/etc/cmqttd/auth` file to be shared with the container as a Docker volume:

```
# mkdir -p /etc/cmqttd
# touch /etc/cmqttd/auth
# chmod 600 /etc/cmqttd/auth
# echo "my-username" >> /etc/cmqttd/auth
# echo "my-password" >> /etc/cmqttd/auth
```

Then to use these authentication details, with TLS enabled:

```
# docker run --device /dev/ttyUSB0 -e "SERIAL_PORT=/dev/ttyUSB0" \
    -e "MQTT_SERVER=192.0.2.1" -e "TZ=Australia/Adelaide" \
    -v /etc/cmqttd:/etc/cmqttd cmqttd
```

If you want to run the `cmqttd` daemon in the background, on the same device as a Home Assistant server with the MQTT broker add-on:

```
# docker run -dit --name cbus --restart=always \
    --device /dev/ttyUSB0 --network hassio \
    -e "TZ=Australia/Adelaide" -e "BROKER_USE_TLS=0" \
    -e "SERIAL_PORT=/dev/ttyUSB0" \
    -e "MQTT_SERVER=core-mosquitto" \
    cmqttd
```

Note: You can verify the hostname of hassio's MQTT broker with: `# docker inspect addon_core_mosquitto`

If you want to run the daemon manually with other settings, you can run `cmqttd` manually within the container (ie: skipping the start-up script) with:

```
# docker run -e "TZ=Australia/Adelaide" cmqttd cmqttd --help
```

Note: When running *without* the start-up script:

- you must write `cmqttd` twice: first as the name of the image, and second as the program inside the image to run.
- none of the environment variables (except *TZ*) are supported – you must use *cmqttd command-line options* instead.
- files in `/etc/cmqttd` are not used unless equivalent *cmqttd command-line options* are manually specified.

CHAPTER 4

Hacking

Information about using the hardware and software.

4.1 Official documentation

You should implement software in conjunction with reading the official documentation. This library attempts to follow its terminology and structures, so understanding what is happening on a lower level is needed particularly when using the lower level interfaces in this library.

There is a large amount of documentation in there that says "these items are deprecated and shouldn't be used". I've noticed that C-Gate and Toolkit will interact with the hardware in these "deprecated" ways...

This doesn't mean implement the library to talk this way. You should implement it properly. Just be aware than when working with implementing a fake PCI or parsing out packets that Clipsal's software generated, be aware they'll do strange and undocumented things.

4.1.1 Geoffry Bennett's reverse engineering notes (2001 - 2004)

Geoffry Bennett gave a talk at a LinuxSA meeting in 2001 and at Linux.conf.au 2004 about his experiences with reverse engineering C-Bus. At the time there was no official protocol documentation available.

The Linux.conf.au 2004 notes cover a lot more information, includes some information about dumping and reverse engineering the contents of NVRAM in units, a Perl client library, and an emulator used for older versions of the Clipsal programming software.

4.2 CNI / network protocol

The C-Bus Toolkit software has a CNI (network) interface mode, which is just the serial protocol over a TCP socket.

Some of the tools here support running in TCP mode with `-t`.

There's also a discovery protocol, however this has not been implemented yet. Patches welcome. :)

4.3 Setting up a fake CNI and sniffing the protocol

If you want to see how Toolkit interacts with a Serial PCI, use the `tcp_serial_redirect.py` script from the pySerial example scripts.

For example:

```
$ python tcp_serial_redirect.py -p /dev/ttyUSB0 -P 10001
```

Congratulations, you now have turned your computer and a 5500PC into a 5500CN without writing a single line of custom code, and saved about 200$. Even a Beaglebone can be had for less than 200$. ;)

Go into Toolkit, set the Default Interface type to "IP Address (CNI)" with the IP and port of the machine running the serial redirector.

You can then use tools like Wireshark to monitor interactions with the C-Bus PCI, instead of using kernel hacks to sniff serial, other redirects, or wiring up your own serial sniffer device. This will aid if you wish to use undocumented commands, or isolate issues in the Clipsal documentation.

You could also use this with tools like C-Gate to get a higher level interface with the C-Bus PCI.

4.4 USB support / 5500PCU

Clipsal's driver is not digitally signed.

It uses `silabser.sys` on Windows, which corresponds to a Silicon Labs CP210X USB-serial bridge. `cbususb.inf` lists the following products:

- `10C4:EA60`: Generic SiLabs CP210X
- `166A:0101`: C-Bus Multi-room Audio Matrix Switcher (560884)
- `166A:0201`: C-Bus Pascal/Programmable Automation Controller (5500PACA)
- `166A:0301`: C-Bus Wireless PC Interface (5800PC). This appears to be an unreleased product.
- `166A:0303`: C-Bus Wired PC Interface (5500PCU)
- `166A:0304`: C-Bus Black & White Touchscreen Mk2 (5000CT2)
- `166A:0305`: C-Bus C-Touch Spectrum Colour Touchscreen (C-5000CT2)
- `166A:0401`: C-Bus Architectural Dimmer (L51xx series)

4.4.1 Linux driver

The `cp210x` kernel module in Linux 2.6.30 and later supports this chipset. However, only the generic adapter and 5500PCU device IDs are included with the kernel for versions before 3.2.22 and 3.5-rc6.

Your distribution vendor may backport the patches in to other kernel versions.

To see which devices your kernel supports, run the following command:

```
$ /sbin/modinfo cp210x | grep v166A
```

If the following is returned, you only have support for the 5500PCU:

```
alias:          usb:v166Ap0303d*dc*dsc*dp*ic*isc*ip*
```

If more lines come back, then your kernel supports all the hardware that is known about at this time.

4.4.2 macOS driver

SiLabs' macOS drivers (v5.2.3) do not list any Clipsal devices in `Info.plist`.

Modifying this file will cause the kext to fail signature verification.

You may be able to use a modified version of this driver if you disable System Integrity Protection from the Recovery OS, but this could have serious repercussions for the security and reliability of your device.

4.5 Unit Tests

Tests use the `unittest` package. To run them:

```
$ python3 -m unittest
```

This targets Python 3.7 and later. Python 2.x are no longer supported.

When implementing a new application, you should copy all of the examples given in the documentation of that application into some tests for that application. Be careful though, there are sometimes errors in Clipsal's documentation, so double check to make sure that the examples are correct. If you find errors in Clipsal's documentation, you should email them about it.

CHAPTER 5

CNI Discovery

At the moment this is a rather unorganised set of notes while I'm still figuring out the protocol.

I've started working on a test program for dissecting the protocol in `experiments/cni_discovery.py`.

5.1 Discovery Query

A client will broadcast a UDP packet on 255.255.255.255:20050.

Data structure is as follows:

```
char[4] command  = "CB 80 00 00"  // CBUS_DISCOVERY_QUERY
char[4] unknown1 = "00 00 00 00"
char[4] unknown2 = "01 01 01 0B"
char[4] unknown3 = "01 1D 80 01"
char[3] unknown4 = "02 47 FF"
```

Example packet:

```
cb:80:00:00:00:00:00:01:01:01:0b:01:1d:80:01:02:47:ff

0xcb, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x01, 0x01, 0x01, 0x0b, 0x01, 0x1d, 0x80, 0x01,
0x02, 0x47, 0xff
```

5.2 Discovery Reply

Replies are sent back to the querying client on port 20050.

Values are in big-endian format (network byte order):

```
char[4]  magic      = 0xcb, 0x81, 0x00, 0x00   CBUS_DISCOVERY_REPLY
char[4]  unknown1   = 0x00, 0x00, 0x00, 0x01   //  0x20, 0xe8, 0xf5, 0x52
char[4]  unknown2   = 0x81, 0x01, 0x00, 0x01
char     product_id = 0x03 // 0x01
char[4]  unknown4   = 0x81, 0x0b, 0x00, 0x02
uint16   port       = 0x27, 0x11  (10001)
char[4]  unknown5   = 0x81, 0x1d, 0x00, 0x01
char     unknown6   = 0x00 // 0x01     (not a flag for "in use")
char[4]  unknown7   = 0x80, 0x01, 0x00, 0x02
char[2]  unknown8   = 0x66, 0x1e // 0x8c, 0x26  (may be a checksum, but doesn't␣
↪appear to be used)
```

Product IDs:

- `01`: CNI2
- `02`: Hidden – Toolkit ignores packets with this product ID. May be used for internal development.
- `03`: WISER
- Other values: “unknown”

Example packet data:

```
Recv

Client 1 (172.26.1.81)
CNI2 port 10001, "not accessible" (controlled by a WISER)

0xcb, 0x81, 0x00, 0x00, 0x20, 0xe8, 0xf5, 0x52,
0x81, 0x01, 0x00, 0x01, 0x01, 0x81, 0x0b, 0x00,
0x02, 0x27, 0x11, 0x81, 0x1d, 0x00, 0x01, 0x01,
0x80, 0x01, 0x00, 0x02, 0x8c, 0x26

Client 2 (172.26.1.80)
WISER port 10001

0xcb, 0x81, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x81, 0x01, 0x00, 0x01, 0x03, 0x81, 0x0b, 0x00,
0x02, 0x27, 0x11, 0x81, 0x1d, 0x00, 0x01, 0x00,
0x80, 0x01, 0x00, 0x02, 0x66, 0x1e

b = "\xcb\x81\x00\x00\x00\x00\x00\x01\x81\x01\x00\x01\x03\x81\x0b\x00\x02
↪'\x11\x81\x1d\x00\x01\x00\x80\x01\x00\x02f" + '\x1e'
```

CHAPTER 6

Wiser

Note: This is an incomplete collection of notes from reverse engineering the Wiser's firmware.

It has not been actively worked on in some years, and the author no longer has access to Wiser hardware.

This library **is not** capable of running on Wiser – and this project's *C-Bus to MQTT bridge* can be used with Home Assistant and entirely replaces the need for Wiser.

It is provided in the hope it could be useful to others, and to serve as a warning against using Wiser hardware. :)

The Wiser is a re-badged SparkLAN WRTR-501 802.11b/g/draft-n WiFi Router with custom firmware. It runs an embedded Linux system, with an expanded web interface for hosting Flash/XMLSocket based control of C-Bus.

According to the source code release from Clipsal, this runs Linux 2.6.17.14. The kernel configuration indicates that the board is a `fv13xx` ARM system. This is also used by:

- Airlink101 AR680W
- PCi MZK-W04N

XMLSocket is also used by the iPhone version of the control software.

Note: In XML outputs in this document, new-line characters and basic formatting whitespace has been added to improve readability. The original data does not contain this, unless otherwise indicated.

6.1 Downloading SWFs

First step is you are directed to the page `/clipsal/resources/wiserui.html`. This in turn loads the SWF `/clipsal/resources/wiserui.swf`.

As this is SWF, there is a cross-domain access policy in place to allow the SWF to connect back to the server on other ports:

```
<cross-domain-policy>
  <allow-access-from domain="*" secure="false" to-ports="8888,8889"/>
</cross-domain-policy>
```

This configuration **disables all cross-domain security** for requests to the Wiser. You could use this to write your own implementation of the Wiser control UI and have it connect back to Wiser's IP. This could also be used to allow any website on the internet you visit to make cross-origin requests to your browser.

The resources and API classes are stored in `/clipsal/resources/resources.swf`. This contains things like the `cbus_controller` class which is used to establish Flash XMLSocket connections.

6.2 Protocol

6.2.1 Discovery and Handshake

After the SWF is started, it loads the configuration file from `/clipsal/resources/local_config.xml`. This looks like:

```
<local_config version="1.0">
  <wiser ip="XXX.XXX.XXX.XXX" port="8888" remote_url="" remote_port="8336"
         remote="0" wan="0"/>
  <client name="Web UI" fullscreen="0" http_auth="0" local_file_access="1"
          local_project="0" local_skin_definition="0"/>
</local_config>
```

Here we see the internal IP address of the Wiser, and the port that is used for XMLSockets requests (`port`). `remote_port` indicates the port used by the CFTP daemon.

6.2.2 Authentication

There is a basic authentication system in place on some of the sockets. This can be established by retrieving the key from `/clipsal/resources/projectorkey.xml`. This file looks like:

```
<cbus_auth_data value="0x12345678"/>
```

This projector key is generated when a project file is first created by PICED. The projector key is **static for all projects created during a particular execution of PICED**.

Rebooting Wiser or changing the HTTP password **never changes this key**. Once someone has this key, they can use it to access Wiser over XMLSocket **in perpetuity**.

The only way to change it is to re-start PICED (if it was already running), create an entirely new project file, and transfer this to the Wiser.

6.2.3 Connecting

There is now enough information to connect to the XMLSocket service on port 8888 of the Wiser (or "port" in `local_config.xml`).

So to start the connection we need to send some commands off to the server to handshake.

This starts with a command called `<cbus_auth_cmd>`. This has three attributes, required **exactly** in this order:

```
<cbus_auth_cmd value="0x12345678" cbc_version="3.7.0" count="0" />
```

- `value` is the value of the `cbus_auth_data` retrieved in the previous step.
- `cbc_version` is the version of the SWF being used. This is found in `wiserui.swf`, in the variable `cbc_version`.
- `count` is the number of times that this session has attempted to authenticate. Set this to 0.

You could also request the project files and skin files in one shot, like this:

```
<cbus_auth_cmd value="0x12345678" cbc_version="3.7.0" count="0" />
<project-file-request />
<skin-file-request />
```

The Wiser responds with a message like this:

```
<ka cbus_connected="1" />
<cbd_version version="Kona_1.24.0" />
<net_status cni_transparent="0" cni="1" cftp="1" cbus="1" ntp="0" />
<cbus_event app="0xdf" name="cbusTimeChanged" time="120103102012.43" dst="0" ntp="0" /
↪>
```

6.2.4 Project and Skin

It also returns a `<Touchscreen>` XML which is a form of the project file, and a `<skin>` XML which contains localised strings and resource image references.

This can also be downloaded from `/clipsal/resources/project.xml` and `/clipsal/resources/skin_definition.xml`, so you can just establish a connection without requesting these files over the XMLSocket. Potentially this could be more reliable.

The project file contains all of the programming in use on the Wiser, button assignments and schedules. It can also contain additional metadata about the installation, if the installer has filled this in.

6.2.5 XMLSocket protocol for dummies

Adobe's documentation describes the XMLSocket protocol as sending XML documents in either direction on the TCP socket, terminated by a null character.

It is like a simple version of WebSockets – client and server may send data at any time, there is no synchronous response mechanism, and very easy to implement.

The XML documents sent do not require the typical XML stanzas at the start of the file specifying encoding, and may also contain multiple top-level (document) elements.

There are third-party client and server libraries available for this protocol.

6.3 Getting a shell

There is console access available via a web interface on the Wiser, using `/console.asp`. It appears to be taken from some Belkin or Linksys reference firmware image.

Redirection of output to a file using > doesn't work correctly in the shell. Regular pipes (`|`) do work.

Only `stdout` is displayed, not `stderr`.

6.3.1 NVRAM

You can dump the NVRAM:

```
$ nvram show
...
wan_proto=dhcp
wan_ipaddr=0.0.0.0
wan_netmask=0.0.0.0
wan_gateway=0.0.0.0
wan_winsserv=
...
```

6.4 CFTP

CFTP is a service which acts as a back-door into the device. It runs on port 8336, and is managed by the service **`cftp_daemon`**.

It has a hard-coded password (`bloop`) to access the service.

Despite the name, it doesn't actually implement FTP – it is used by Clipsal's programming software in order to manage the device. It appears to have the following functionality:

- Manage port forwards inside of the network when the device is acting as the router for the network. Unknown how this is controlled.
- Reflash the contents of partition 6 of FLASH (label: `clipsal`). Appears to be a gzip-compressed tarball, which gets extracted to `/www/clipsal/resources`.

Communication with the server is done with a simple text-based protocol, with the UNIX newline character indicating the end of command. DOS or other style line feeds do not work.

If the daemon does not understand your command, it will simply send no response.

6.4.1 Startup process

On startup, the process will:

1. Delete `/tmp/*.tar.gz`.
2. Copy the contents of `/dev/mtblock/6` to `/tmp/test.cta`.
3. Mount a new ramfs to `/www/clipsal/resources/`
4. Extract `settings.conf` from the gzip-compressed tarball `/tmp/test.cta` to `/www/clipsal/resources/`.
5. Read daemon configuration from `settings.conf`.
6. Extract all files from the tarball to `/www/clipsal/resources/`.

6.4.2 Unauthenticated state

Connecting to the service yields a welcome message:

```
200 Welcome
```

PASS

Client command:

```
PASS bloop
```

The server will respond that you are logged in successfully, and transition your connection to the authenticated state:

```
201 Logged in
```

Note: There is no way to change this password. It is hard coded in Wiser's firmware.

Sending other passwords yield no response.

6.4.3 Authenticated state

When in the authenticated state, the network code appears to be far less robust. Sending large commands causes the daemon to crash.

This may be an effective and easy way to disable **`cftp_daemon`** on the device.

PASS

Client command:

```
PASS bloop
```

Server response:

```
201 Logged in
```

Transitions to the authenticated state. Has no effect in authenticated mode.

Note: There is no way to change this password. It is hard coded in Wiser's firmware.

Sending other passwords yield no response.

VERINFO

Client command:

```
VERINFO
```

Server response:

```
202-HomeGateVersion=4.0.41.0
202-CTCServerVersion=Kona_1.24.0
202-UnitName=EXAMPLE
202 WindowsOSVersion=5.1.2600 Service Pack 2
```

Retrieves information about the version of CFTP running on the Wiser, and the C-Bus network's project name.

The `WindowsOSVersion` information is a hard-coded string.

HGSTATUS

Client command:

```
HGSTATUS
```

Server response:

```
202-HGRUNNING=False
202-HGLOGGING=False
202 CURRPROJ=C:\HomeGate\Projects\Current\EXAMPLEproj.tar.gz
```

Retrieves the current project name running on the Wiser, and status of "HG"? This is hard coded to always return False to both HGRUNNING and HGLOGGING.

The path is faked by the daemon, with "EXAMPLE" replaced by the project name.

GETFILELIST

Client command:

```
GETFILELIST
```

Server response:

```
202 FILE1=C:\HomeGate\Projects\Current\EXAMPLEproj.tar.gz
```

Retrieves a list of "files" on the device associated with the project. This only returns the project file.

The path is faked by the daemon, with "EXAMPLE" replaced by the project name.

GETPROJ

Client command:

```
GETPROJ
```

Server response:

```
202-Port=8337
202 FILE=C:\HomeGate\Projects\Current\EXAMPLEproj.tar.gz
```

Returns the "project filename" for the contents of flash partition 6. The path information is hard coded and fake, with "EXAMPLE" replaced by the project name.

INSTALL

Client command:

```
INSTALL PROJECT example.tar.gz
```

Server response:

```
202 Port=8337
```

Starts an out of band transfer for overwriting the Wiser's project file.

The server opens up another TCP server on a different port (on Wiser, this is always 8337) in order to accept the file transfer out of band.

6.4.4 Project file transfer

Project file transfer is done on another port (always 8337), and initiated by the `INSTALL` command.

The client immediately sends:

```
FILE example.tar.gz
```

This is then immediately followed by a UNIX newline character, and then the file length as a 32-bit unsigned big-endian integer.

Files must not be bigger than 512kB, or the transfer will be rejected by the Wiser. File names must end in `.tar.gz`.

Projects must also not extract to a size greater than about 1 MiB: Wiser stores the contents of this archive in ramfs, so larger archives will use all available RAM on the Wiser, and cannot be freed, leading to Linux's oomkiller to run or processes to fail to dynamically allocate memory. This has the potential in turn to partially brick the Wiser – **`cftp_daemon`** will not be able to copy a new project file into RAM temporarily for flashing, and may be permanently stuck in this state. This partial brick state could probably gotten around by writing NULL over the contents of `/dev/mtdblock/6`, then transferring a new project file.

6.5 Firmware image

Firmware image for the device is bundled with the PICED software as `Firmware/firmware_1_24_0.img`.

The tool binwalk shows the layout of the firmware image:

```
0x13        uImage header, header size: 64 bytes, header CRC: 0x2781C02C,
            created: Mon Oct  3 11:26:33 2011, image size: 722439 bytes,
            Data Address: 0x40008000, Entry Point: 0x40008000,
            data CRC: 0xF7547123, OS: Linux, CPU: ARM,
            image type: OS Kernel Image, compression type: lzma,
            image name: Linux-2.6.17

0x53        LZMA compressed data, properties: 0x5D,
            dictionary size: 8388608 bytes, uncompressed size: 2015280 bytes

0xC0013     Squashfs filesystem, little endian, version 2.1,
            size: 1736392 bytes, 435 inodes, blocksize: 65536 bytes,
            created: Mon Oct  3 11:27:23 2011
```

Appears to be a uBoot image with some extra headers on the image.

6.5.1 Extracting root filesystem

> **Warning:** The links in this section are broken as Google Code has shut down.

The version of squashfs used by the root filesystem is very old, and current Linux kernels are incapable of mounting it. It requires an LZMA version of squashfs-2.1 in order to extract it, available from firmware-mod-kit. Their SVN repository contains all the components needed:

```
$ svn co https://firmware-mod-kit.googlecode.com/svn/trunk/src/lzma/
$ svn co https://firmware-mod-kit.googlecode.com/svn/trunk/src/squashfs-2.1-r2/
$ cd squashfs-2.1-r2
$ make
```

Once built, extract the root filesystem with:

```
$ binwalk -D squashfs:squashfs firmware_1_24_0.img
$ ./squashfs-2.1-r2/unsquashfs-lzma C0013.squashfs
```

This will then give an extracted copy of the root filesystem in the directory `squashfs-root`.

6.5.2 Filesystem observations

These are things that need some more investigation:

- NTP client which has 32 hard-coded NTP server IP addresses.

CHAPTER 7

dump_labels utility

dump_labels parses a Toolkit backup file (CBZ) and prints out group and unit address labels into a JSON file.

It will remove all unneeded programming markup from the CBZ, and leave a skeleton of information which can be used in conjunction with the library and other applications.

7.1 Invocation

CHAPTER 8

libcbus module index

8.1 cbus Package

This is the package where all C-Bus modules are defined.

8.1.1 Common functions

cbus.common defines various common helper utilities used by the library, and constants required to communicate with the C-Bus network.

The majority of the functionality shouldn't be needed by your own application, however it is used internally within the protocol encoders and decoders.

class cbus.common.**Application**

Bases: enum.IntEnum

An enumeration.

CLOCK = 223

ENABLE = 203

LIGHTING = 56

LIGHTING_30 = 48

LIGHTING_31 = 49

LIGHTING_32 = 50

LIGHTING_33 = 51

LIGHTING_34 = 52

LIGHTING_35 = 53

LIGHTING_36 = 54

```
LIGHTING_37 = 55
LIGHTING_38 = 56
LIGHTING_39 = 57
LIGHTING_3a = 58
LIGHTING_3b = 59
LIGHTING_3c = 60
LIGHTING_3d = 61
LIGHTING_3e = 62
LIGHTING_3f = 63
LIGHTING_40 = 64
LIGHTING_41 = 65
LIGHTING_42 = 66
LIGHTING_43 = 67
LIGHTING_44 = 68
LIGHTING_45 = 69
LIGHTING_46 = 70
LIGHTING_47 = 71
LIGHTING_48 = 72
LIGHTING_49 = 73
LIGHTING_4a = 74
LIGHTING_4b = 75
LIGHTING_4c = 76
LIGHTING_4d = 77
LIGHTING_4e = 78
LIGHTING_4f = 79
LIGHTING_50 = 80
LIGHTING_51 = 81
LIGHTING_52 = 82
LIGHTING_53 = 83
LIGHTING_54 = 84
LIGHTING_55 = 85
LIGHTING_56 = 86
LIGHTING_57 = 87
LIGHTING_58 = 88
LIGHTING_59 = 89
LIGHTING_5a = 90
```

LIGHTING_5b = 91

LIGHTING_5c = 92

LIGHTING_5d = 93

LIGHTING_5e = 94

LIGHTING_5f = 95

LIGHTING_FIRST = 48

LIGHTING_LAST = 95

MASTER_APPLICATION = 255

STATUS_REQUEST = 255

TEMPERATURE = 25

class cbus.common.**CAL**
Bases: enum.IntEnum

An enumeration.

ACKNOWLEDGE = 50

EXTENDED_STATUS = 224

GET_STATUS = 42

IDENTIFY = 33

RECALL = 26

REPLY = 128

RESET = 8

STANDARD_STATUS = 192

class cbus.common.**ClockAttribute**
Bases: enum.IntEnum

An enumeration.

DATE = 2

TIME = 1

class cbus.common.**ClockCommand**
Bases: enum.IntEnum

An enumeration.

REQUEST_REFRESH = 17

UPDATE_NETWORK_VARIABLE = 8

class cbus.common.**DestinationAddressType**
Bases: enum.IntEnum

Destination Address Type (DAT).

Ref: Serial Interface Guide, s3.4. Other values reserved.

POINT_TO_MULTIPOINT = 5

POINT_TO_POINT = 6

POINT_TO_POINT_TO_MULTIPOINT = 3

UNSET = 0

class cbus.common.**EnableCommand**
Bases: enum.IntEnum

An enumeration.

SET_NETWORK_VARIABLE = 2

class cbus.common.**ExtendedCALType**
Bases: enum.IntEnum

An enumeration.

BINARY = 0

LEVEL = 7

class cbus.common.**GroupState**
Bases: enum.IntEnum

An enumeration.

ERROR = 3

MISSING = 0

OFF = 2

ON = 1

class cbus.common.**IdentifyAttribute**
Bases: enum.IntEnum

IDENTIFY attributes.

See Serial Interface Guide, s7.2.

CUR_LVL = 15

DELAYS = 12

DSI_STATUS = 17

EXTENDED = 4

FIRMWARE_VER = 2

GAV_CURRENT = 8

GAV_PHY_ADDR = 10

GAV_STORED = 9

LOGIC_ASSIGN = 11

MANUFACTURER = 0

MAX_LVL = 14

MIN_LVL = 13

NET_TERM_LVL = 5

NET_VOLTAGE = 7

OUT_SUMMARY = 16

SUMMARY = 3

TERM_LVL = 6

TYPE = 1

class cbus.common.**LightCommand**
Bases: enum.IntEnum

An enumeration.

LIGHT_LABEL = 160

OFF = 1

ON = 121

RAMP_00_04 = 10

RAMP_00_08 = 18

RAMP_00_12 = 26

RAMP_00_20 = 34

RAMP_00_30 = 42

RAMP_00_40 = 50

RAMP_01_00 = 58

RAMP_01_30 = 66

RAMP_02_00 = 74

RAMP_03_00 = 82

RAMP_05_00 = 90

RAMP_07_00 = 98

RAMP_10_00 = 106

RAMP_15_00 = 114

RAMP_17_00 = 122

RAMP_FASTEST = 2

RAMP_INSTANT = 2

RAMP_SLOWEST = 122

TERMINATE_RAMP = 9

class cbus.common.**PriorityClass**
Bases: enum.IntEnum

An enumeration.

CLASS_1 = 3

CLASS_2 = 2

CLASS_3 = 1

CLASS_4 = 0

cbus.common.**add_cbus_checksum**(*i: bytes*) → bytes
Appends a C-Bus checksum to a given message.

Parameters **i** (*bytes*) – The C-Bus message to append a checksum to. Must not be in base16 format.

Returns The C-Bus message with the checksum appended to it.

Return type bytes

cbus.common.**cbus_checksum**(*i: bytes*) → int
Calculates the checksum of a C-Bus command string.

Fun fact: C-Bus toolkit and C-Gate do not use commands with checksums.

Parameters **i** (*bytes*) – The C-Bus data to calculate the checksum of. Must not be in base16 format.

Returns The checksum value of the given input

cbus.common.**check_ga**(*group_addr: int*) → None
Validates a given group address, throwing ValueError if not.

Parameters **group_addr** – Input group address to validate.

Raises **ValueError** – If group address is invalid

cbus.common.**duration_to_ramp_rate**(*seconds: int*) → cbus.common.LightCommand
Converts a given duration into a ramp rate code.

Parameters **seconds** (*int*) – The number of seconds that the ramp is over.

Returns The ramp rate code for the duration given.

Return type int

cbus.common.**get_real_cbus_checksum**(*i: bytes*) → int
Calculates the current C-Bus checksum for a given message which already has a checksum appended to it.

Parameters **i** – The C-Bus message to generate an actual checksum for, in raw format.

cbus.common.**ramp_rate_to_duration**(*rate: int*) → int
Converts a given ramp rate code into a duration in seconds.

Parameters **rate** (*int*) – The ramp rate code to convert.

Returns The number of seconds the ramp runs over.

Return type int

Raises **KeyError** – If the given ramp rate code is invalid.

cbus.common.**validate_cbus_checksum**(*i: bytes*) → bool
Verifies a C-Bus checksum from a given message.

Parameters **i** – The C-Bus message to verify the checksum of, in raw format.

Returns True if the checksum is correct, False otherwise.

cbus.common.**validate_ga**(*group_addr: int*) → bool
Validates a given group address.

Parameters **group_addr** – Input group address to validate.

Returns True if the given group address is valid, False otherwise.

8.1.2 Protocol

C-Bus uses it's own protocol in order to send messages over the C-Bus PHY.

This is reflected in the PC Interface protocol.

This package contains classes needed in order to operate with the protocol.

Note: The only "stable" API for this project is the *C-Bus to MQTT bridge*, when accessed via an MQTT broker.

The lower level APIs are subject to change without notice, as we learn new information about the C-Bus control protocol, and functionality from other applications is brought into the *C-Bus to MQTT bridge*.

`base_packet`: Base Packet

class cbus.protocol.base_packet.**BasePacket**(*checksum: bool = True*, *destination_address_type: cbus.common.DestinationAddressType = <DestinationAddressType.UNSET: 0>*, *rc: int = 0*, *dp: bool = False*, *priority_class: cbus.common.PriorityClass = <PriorityClass.CLASS_4: 0>*)

Bases: abc.ABC

encode() → bytes

encode_packet() → bytes

flags

class cbus.protocol.base_packet.**InvalidPacket**(*payload: bytes*, *exception: Optional[Exception] = None*)

Bases: cbus.protocol.base_packet._SpecialPacket

Invalid packet data.

encode()

exception = None

class cbus.protocol.base_packet.**SpecialClientPacket**

Bases: cbus.protocol.base_packet._SpecialPacket, abc.ABC

Client -> PCI communications have some special packets, which we make subclasses of SpecialClientPacket to make them entirely separate from normal packets.

These have non-standard methods for serialisation.

class cbus.protocol.base_packet.**SpecialServerPacket**

Bases: cbus.protocol.base_packet._SpecialPacket, abc.ABC

PCI -> Client has some special packets that we make subclasses of this, because they're different to regular packets.

These have non-standard serialisation methods.

dm_packet: Device Management Packet

class cbus.protocol.dm_packet.**DeviceManagementPacket**(*checksum: bool = True*, *priority_class: cbus.common.PriorityClass = <PriorityClass.CLASS_2: 2>*, *parameter: int = 0*, *value: int = 0*)

Bases: *cbus.protocol.base_packet.BasePacket*

static decode_packet(*data: bytes*, *checksum: bool*, *priority_class: cbus.common.PriorityClass*) → cbus.protocol.dm_packet.DeviceManagementPacket

encode() → bytes

packet Module

cbus.protocol.packet.**decode_packet**(*data: bytes*, *checksum: bool = True*, *strict: bool = True*, *from_pci: bool = True*) → Tuple[Union[cbus.protocol.base_packet.BasePacket, cbus.protocol.cal.extended.ExtendedCAL, cbus.protocol.cal.identify.IdentifyCAL, cbus.protocol.cal.reply.ReplyCAL, cbus.protocol.cal.recall.RecallCAL, None], int]

Decodes a single C-Bus Serial Interface packet.

The return value is a tuple:

0. The packet that was parsed, or None if there was no packet that could be parsed.
1. The buffer position that we parsed up to. This may be non-zero even if the packet was None (eg: Cancel request).

Note: this decoder does not support unaddressed packets (such as Standard Format Status Replies).

Note: Direct Command Access causes this method to return AnyCAL instead of a BasePacket.

Parameters

- **data** – The data to parse, in encapsulated serial format.
- **checksum** – If True, requires a checksum for all packets
- **strict** – If True, returns InvalidPacket whenever checksum is incorrect. Otherwise, only emits a warning.
- **from_pci** – If True, parses the packet as if it were sent from/by a PCI – if your software was sent packets by a PCI, this is what you want.

  If False, this parses the packet as if it were sent to a PCI; parsing messages that software expecting to communicate with a PCI sends. This could be used to build a fake PCI, or analyse the behaviour of other C-Bus software.

cbus.protocol.packet.**int2byte**()

S.pack(v1, v2, ...) -> bytes

Return a bytes object containing values v1, v2, ... packed according to the format string S.format. See help(struct) for more on format strings.

pm_packet Module

class cbus.protocol.pm_packet.**PointToMultipointPacket**(*checksum: bool = True, priority_class: cbus.common.PriorityClass = <PriorityClass.CLASS_4: 0>, application: Optional[cbus.common.Application] = None, sals: Union[cbus.protocol.application.sal.SAL, Sequence[cbus.protocol.application.sal.SAL], None] = None*)

Bases: *cbus.protocol.base_packet.BasePacket*, collections.abc.Sequence, typing.Generic

Point to Multipoint Packet

Ref: Serial Interface User Guide, s4.2.9.2

append_sal(*sal: cbus.protocol.application.sal.SAL*) → None

clear_sal() → None
Removes all SALs from this packet.

classmethod decode_packet(*data: bytes, checksum: bool, priority_class: cbus.common.PriorityClass*) → cbus.protocol.pm_packet.PointToMultipointPacket

encode()

index(*x: cbus.protocol.application.sal.SAL, start: int = Ellipsis, end: int = Ellipsis*) → int
Finds a SAL within this packet.

Raises **ValueError** – if not present

pp_packet Module

class cbus.protocol.pp_packet.**PointToPointPacket**(*checksum: bool = True, priority_class: cbus.common.PriorityClass = <PriorityClass.CLASS_4: 0>, unit_address: int = 0, bridge_address: int = 0, hops: Optional[Sequence[int]] = None, cals: Optional[Sequence[Union[cbus.protocol.cal.extended.ExtendedC cbus.protocol.cal.identify.IdentifyCAL, cbus.protocol.cal.reply.ReplyCAL, cbus.protocol.cal.recall.RecallCAL]]] = None*)

Bases: *cbus.protocol.base_packet.BasePacket*, collections.abc.Sequence, typing.Generic

classmethod decode_cal(*data: bytes*) → Tuple[Union[cbus.protocol.cal.extended.ExtendedCAL, cbus.protocol.cal.identify.IdentifyCAL, cbus.protocol.cal.reply.ReplyCAL, cbus.protocol.cal.recall.RecallCAL], int]

classmethod decode_packet(*data: bytes, checksum: bool, priority_class: cbus.common.PriorityClass*) → cbus.protocol.pp_packet.PointToPointPacket

encode() → bytes

index(*x: Union[cbus.protocol.cal.extended.ExtendedCAL, cbus.protocol.cal.identify.IdentifyCAL, cbus.protocol.cal.reply.ReplyCAL, cbus.protocol.cal.recall.RecallCAL], start: int = Ellipsis, end: int = Ellipsis*) → int

Finds a CAL within this packet.

Raises **ValueError** – if not present

reset_packet: PCI Reset Packet

class cbus.protocol.reset_packet.**ResetPacket**

Bases: *cbus.protocol.base_packet.SpecialClientPacket*

encode()

scs_packet Module

class cbus.protocol.scs_packet.**SmartConnectShortcutPacket**

Bases: *cbus.protocol.base_packet.SpecialClientPacket*

encode() → bytes

pciprotocol Module

class cbus.protocol.pciprotocol.**PCIProtocol**(*timesync_frequency: int = 10, handle_clock_requests: bool = True, connection_lost_future: Optional[_asyncio.Future] = None*)

Bases: cbus.protocol.cbus_protocol.CBusProtocol

Implements an asyncio Protocol for communicating with a C-Bus PCI/CNI over TCP or serial.

clock_datetime(*when: Optional[datetime.datetime] = None*)

Sends the system's local time to the CBus network.

Parameters **when** (*datetime.datetime*) – The time and date to send to the CBus network. Defaults to current local time.

connection_lost(*exc: Optional[Exception]*) → None

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made(*transport: asyncio.transports.WriteTransport*) → None

Called by asyncio when a connection is made to the PCI. This will perform a reset of the PCI to establish the correct communications protocol, and start time synchronisation.

handle_cbus_packet(*p: cbus.protocol.base_packet.BasePacket*) → None

Dispatches all packet types into a high level event handler.

identify(*unit_address*, *attribute*)

Sends an IDENTIFY command to the given unit_address.

Parameters

- **unit_address** (*int*) – Unit address to send the packet to
- **attribute** (*int*) – Attribute ID to retrieve information for. See s7.2 of Serial Interface Guide for acceptable codes.

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_off(*group_addr: Union[int, Iterable[int]]*)
Turns off the lights for the given group_id.

Parameters **group_addr** (*int, or iterable of ints of length <= 9.*) – Group address(es) to turn the lights off for, up to 9

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_on(*group_addr: Union[int, Iterable[int]]*)
Turns on the lights for the given group_id.

Parameters **group_addr** (*int, or iterable of ints of length <= 9.*) – Group address(es) to turn the lights on for, up to 9

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_ramp(*group_addr: int*, *duration: int*, *level: int = 255*)
Ramps (fades) a group address to a specified lighting level.

Note: CBus only supports a limited number of fade durations, in decreasing accuracy up to 17 minutes (1020 seconds). Durations longer than this will throw an error.

A duration of 0 will ramp "instantly" to the given level.

Parameters

- **group_addr** (*int*) – The group address to ramp.
- **duration** (*int*) – Duration, in seconds, that the ramp should occur over.
- **level** (*int*) – A value between 0 and 255 indicating the brightness.

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_terminate_ramp(*group_addr: Union[int, Iterable[int]]*)
Stops ramping a group address at the current point.

Parameters **group_addr** (*int*) – Group address to stop ramping of.

Returns Single-byte string with code for the confirmation event.

Return type string

on_clock_request(*source_addr*)
Event called when a unit requests time from the network.

Parameters **source_addr** (*int*) – Source address of the unit requesting time.

on_clock_update(*source_addr*, *val*)
Event called when a unit sends time to the network.

Parameters **source_addr** (*int*) – Source address of the unit requesting time.

on_confirmation(*code: bytes*, *success: bool*)
Event called when a command confirmation event was received.

Parameters

- **code** – A single byte matching the command that this is a response to.
- **success** – True if the command was successful, False otherwise.

on_lighting_group_off(*source_addr: int*, *group_addr: int*)
Event called when a lighting application "off" request is received.

Parameters

- **source_addr** (*int*) – Source address of the unit that generated this event.
- **group_addr** (*int*) – Group address being turned off.

on_lighting_group_on(*source_addr: int*, *group_addr: int*)
Event called when a lighting application "on" request is received.

Parameters

- **source_addr** (*int*) – Source address of the unit that generated this event.
- **group_addr** (*int*) – Group address being turned on.

on_lighting_group_ramp(*source_addr: int*, *group_addr: int*, *duration: int*, *level: int*)
Event called when a lighting application ramp (fade) request is received.

Parameters

- **source_addr** (*int*) – Source address of the unit that generated this event.
- **group_addr** (*int*) – Group address being ramped.
- **duration** (*int*) – Duration, in seconds, that the ramp is occurring over.
- **level** (*int*) – Target brightness of the ramp (0 - 255).

on_lighting_group_terminate_ramp(*source_addr: int*, *group_addr: int*)
Event called when a lighting application "terminate ramp" request is received.

Parameters

- **source_addr** (*int*) – Source address of the unit that generated this event.
- **group_addr** (*int*) – Group address stopping ramping.

on_lighting_label_text(*source_addr: int*, *group_addr: int*, *flavour: int*, *language_code: int*, *label: str*)
Event called when a group address' label text is updated.

Parameters

- **source_addr** (*int*) – Source address of the unit that generated this event.
- **group_addr** (*int*) – Group address to relabel.
- **flavour** (*int*) – "Flavour" of the label to update. This is a value between 0 and 3.
- **language_code** (*int*) – Language code for the label.
- **label** (*str*) – Label text, or an empty string to delete the label.

on_mmi(*application: int*, *data: bytes*)
Event called when a MMI was received.

Parameters

- **application** – Application that this MMI concerns.
- **data** – MMI data

on_pci_cannot_accept_data()
Event called whenever the PCI cannot accept the supplied data. Common reasons for this occurring:

- The checksum is incorrect.
- The buffer in the PCI is full.

Unfortunately the PCI does not tell us which requests these are associated with.

This error can occur if data is being sent to the PCI too quickly, or if the cable connecting the PCI to the computer is faulty.

While the PCI can operate at 9600 baud, this only applies to data it sends, not to data it recieves.

on_pci_power_up()
If Power-up Notification (PUN) is enabled on the PCI, this event is fired.

This event may be fired multiple times in quick succession, as the PCI will send the event twice.

on_reset()
Event called when the PCI has been hard reset.

pci_reset()
Performs a full reset of the PCI.

timesync()

pciserverprotocol Module

class cbus.protocol.pciserverprotocol.**PCIServerProtocol**
Bases: cbus.protocol.cbus_protocol.CBusProtocol

Implements an asyncio Protocol for simulating a C-Bus PCI/CNI over TCP or serial.

This presently only implements a subset of the protocol used by PCIProtocol.

connection_made(*transport*)
Called by asyncio a connection is made to the simulated PCI.

This doesn't get fired in normal serial connections, however we'll send a power up notification (PUN).

Serial Interface User Guide s4.3.3.4, page 33

echo(*data: bytes*) → None
Called when data needs to be echoed to the underlying transport.

This is only called when running in server mode.

The default implementation is a stub, and needs to be implemented when running in server mode. This should only do something if the virtual PCI is in `basic` mode.

handle_cbus_packet(*p: cbus.protocol.base_packet.BasePacket*) → None
Handles a single CBus packet.

lighting_group_off(*source_addr*, *group_addr*)
Turns off the lights for the given group_addr.

Parameters

- **source_addr** (*int*) – Source address of the event.

- **group_addr** (*int*) – Group address to turn the lights on for.

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_on(*source_addr*, *group_addr*)
Turns on the lights for the given group_addr.

Parameters

- **source_addr** (*int*) – Source address of the event.
- **group_addr** (*int*) – Group address to turn the lights on for.

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_ramp(*source_addr*, *group_addr*, *duration*, *level=1.0*)
Ramps (fades) a group address to a specified lighting level.

Note: CBus only supports a limited number of fade durations, in decreasing accuracy up to 17 minutes (1020 seconds). Durations longer than this will throw an error.

A duration of 0 will ramp "instantly" to the given level.

Parameters

- **source_addr** (*int*) – Source address of the event.
- **group_addr** (*int*) – The group address to ramp.
- **duration** (*int*) – Duration, in seconds, that the ramp should occur over.
- **level** (*float*) – An amount between 0.0 and 1.0 indicating the brightness to set.

Returns Single-byte string with code for the confirmation event.

Return type string

lighting_group_terminate_ramp(*source_addr*, *group_addr*)
Stops ramping a group address at the current point.

Parameters

- **source_addr** (*int*) – Source address of the event.
- **group_addr** (*int*) – Group address to stop ramping of.

Returns Single-byte string with code for the confirmation event.

Return type string

on_clock_request()
Event called when a clock application "request time" is recieved.

on_clock_update(*val*)
Event called when a clock application "update time" is recieved.

Parameters

- **variable** (*datetime.date or datetime.time*) – Clock variable to update.
- **val** – Clock value

on_lighting_group_off(*group_addr*)
Event called when a lighting application "off" request is recieved.

Parameters **group_addr** (*int*) – Group address being turned off.

on_lighting_group_on(*group_addr*)
Event called when a lighting application "on" request is recieved.

Parameters **group_addr** (*int*) – Group address being turned on.

on_lighting_group_ramp(*group_addr*, *duration*, *level*)
Event called when a lighting application ramp (fade) request is recieved.

Parameters

- **group_addr** (*int*) – Group address being ramped.
- **duration** (*int*) – Duration, in seconds, that the ramp is occurring over.
- **level** (*float*) – Target brightness of the ramp (0.0 - 1.0).

on_lighting_group_terminate_ramp(*group_addr*)
Event called when a lighting application "terminate ramp" request is recieved.

Parameters **group_addr** (*int*) – Group address ramp being terminated.

on_master_application_status(*group_address: int*) → None
Event for Status Request for the master application.

This expects a binary status report of the presence of every unit on the network. :param group_address: Group number to start from

on_reset()
Event called when the PCI has been hard reset.

send_confirmation(*code: bytes*, *ok: bool = True*)

send_error()

Applications

Running ontop of the C-Bus protocols are applications.

This package provides encoders and decoders for application-level messages on the C-Bus network.

Application messages inside of C-Bus packets are called "Specific Application Language", or SALs for short. A packet may contain many SALs for a single application, up to the MTU of the C-Bus network.

Clock and Timekeeping Application

The Clock and Timekeeping Application is used to provide access to date and time information to CBus units.

This is used for example in conjunction with programmable units that act differently depending on the time or date, and with touchscreen units that may display the time on their screens.

Please refer to this document in conjunction with the Clock and Timekeeping Application Guide published by Clipsal.

class cbus.protocol.application.clock.**ClockApplication**
Bases: cbus.protocol.application.sal.BaseApplication

This class is called in the cbus.protocol.applications.APPLICATIONS dict in order to describe how to decode clock and timekeeping application events received from the network.

Do not call this class directly.

static decode_sals(*data: bytes*) → Sequence[cbus.protocol.application.sal.SAL]
Decodes a clock and timekeeping application packet and returns its SAL(s).

static supported_applications() → Set[cbus.common.Application]
Gets a list of supported Application IDs for the application.

All application IDs must be in the range 0x00 - 0xff.

class cbus.protocol.application.clock.**ClockSAL**
Bases: cbus.protocol.application.sal.SAL, abc.ABC

Base type for clock and timekeeping application SALs.

application

static decode_sals(*data: bytes*) → Sequence[cbus.protocol.application.clock.ClockSAL]
Decodes a clock broadcast application packet and returns it's SAL(s).

Parameters **data** – SAL data to be parsed.

Returns The SAL messages contained within the given data.

Return type list of cbus.protocol.application.clock.ClockSAL

class cbus.protocol.application.clock.**ClockUpdateSAL**(*val: Union[datetime.date, datetime.time]*)
Bases: *cbus.protocol.application.clock.ClockSAL*

Clock update event SAL.

Informs the network of the current time.

Creates a new SAL Clock update message.

Use `clock_update_sal(val)` instead of this constructor, as that method handles `datetime.datetime` objects (in addition to `datetime.date` and `datetime.time`, always returns `Sequence[ClockUpdateSAL]`.

Parameters **val** – The value of that variable. Dates are represented in native date format, and times are represented in native time format.

classmethod decode(*data: bytes*, *command_code: int*) → Tuple[Optional[cbus.protocol.application.clock.ClockSAL], bytes]
Do not call this method directly – use ClockSAL.decode

encode() → bytes

is_date

is_time

class cbus.protocol.application.clock.**ClockRequestSAL**
Bases: *cbus.protocol.application.clock.ClockSAL*

Clock request event SAL.

Requests network time.

Creates a new SAL Clock request message.

classmethod decode(*data: bytes*) → Tuple[Optional[cbus.protocol.application.clock.ClockSAL], bytes]
Do not call this method directly – use ClockSAL.decode

encode() → bytes

cbus.protocol.application.clock.**clock_update_sal**(*val: Union[datetime.date, datetime.time, datetime.datetime]*) → Sequence[cbus.protocol.application.clock.ClockUpdateSAL]

Creates Clock Update SAL(s) based on Python datetime objects.

Parameters **val** – The value to set in the `ClockUpdateSAL`. If this is a `datetime.datetime`, this will create multiple `ClockUpdateSAL` objects. If this is a `datetime.date` or `datetime.time`, this will only create only a single `ClockUpdateSAL`.

Returns Sequence of `ClockUpdateSAL`, regardless of input value type.

Raises **TypeError** – On invalid input type.

Enable Control Application

The Enable Control Application is used to set network variables on CBus units.

This can change the behaviour of certain elements of the network, or allow some reprogramming of devices.

Please refer to this document in conjunction with the Enable Control Application Guide published by Clipsal.

class cbus.protocol.application.enable.**EnableApplication**

Bases: `cbus.protocol.application.sal.BaseApplication`

This class is called in the cbus.protocol.applications.APPLICATIONS dict in order to describe how to decode enable broadcast application events received from the network.

Do not call this class directly.

classmethod **decode_sals**(*data: bytes*) → List[cbus.protocol.application.enable.EnableSAL]

Decodes a enable broadcast application packet and returns its SAL(s).

static **supported_applications**() → Set[cbus.common.Application]

Gets a list of supported Application IDs for the application.

All application IDs must be in the range 0x00 - 0xff.

class cbus.protocol.application.enable.**EnableSAL**

Bases: `cbus.protocol.application.sal.SAL`

Base type for enable control application SALs.

application

static **decode_sals**(*data: bytes*) → List[cbus.protocol.application.enable.EnableSAL]

Decodes a enable control application packet and returns it's SAL(s).

Parameters **data** (*str*) – SAL data to be parsed.

Returns The SAL messages contained within the given data.

Return type list of cbus.protocol.application.enable.EnableSAL

class cbus.protocol.application.enable.**EnableSetNetworkVariableSAL**(*variable*, *value*)

Bases: *cbus.protocol.application.enable.EnableSAL*

Enable control Set Network Variable SAL.

Sets a network variable.

Creates a new SAL Enable Control Set Network Variable

Parameters

- **variable** (*int*) – The variable ID being changed
- **value** (*int*) – The value of the network variable

classmethod decode(*data*)
Do not call this method directly – use EnableSAL.decode

encode() → bytes

Lighting Application

The lighting application is the most commonly used application on the C-Bus network.

It is used for turning lights on and off, and setting lights to a particular brightness.

Sometimes the lighting application is used to control other, non-lighting loads, such as exhaust fans.

Please refer to this document in conjunction with the Lighting Application Guide published by Clipsal.

class cbus.protocol.application.lighting.**LightingApplication**
Bases: cbus.protocol.application.sal.BaseApplication

This class is called in the cbus.protocol.applications.APPLICATIONS dict in order to describe how to decode lighting application events recieved from the network.

Do not call this class directly.

static decode_sals(*data: bytes*) → List[cbus.protocol.application.sal.SAL]
Decodes a SAL message

static supported_applications() → FrozenSet[int]
Gets a list of supported Application IDs for the application.

All application IDs must be in the range 0x00 - 0xff.

class cbus.protocol.application.lighting.**LightingSAL**(*group_address: int*)
Bases: cbus.protocol.application.sal.SAL, abc.ABC

Base type for lighting application SALs.

This should not be called directly by your code!

Use one of the subclasses of cbus.protocol.lighting.LightingSAL instead.

application

static decode(*data: bytes*, *command_code: int*, *group_address: int*) → Tuple[Optional[cbus.protocol.application.lighting.LightingSAL], bytes]

static decode_sals(*data: bytes*) → List[cbus.protocol.application.lighting.LightingSAL]
Decodes a lighting application packet and returns it's SAL(s).

Parameters **data** – SAL data to be parsed.

Returns The SAL messages contained within the given data.

Return type list of cbus.protocol.application.lighting.LightingSAL

encode() → bytes
Encodes the SAL into a format for sending over the C-Bus network.

class cbus.protocol.application.lighting.**LightingRampSAL**(*group_address: int*, *duration: int*, *level: int*)
Bases: *cbus.protocol.application.lighting.LightingSAL*

Lighting Ramp (fade) event SAL

Instructs the given group address to fade to a lighting level (brightness) over a given duration.

Creates a new SAL Lighting Ramp message.

Parameters

- **group_address** (*int*) – The group address to ramp.
- **duration** (*int*) – The duration to ramp over, in seconds.
- **level** (*int*) – The level to ramp to, with 0 indicating off, and 255 indicating full brightness.

static decode(*data: bytes*, *command_code: int*, *group_address: int*) → Tuple[Optional[cbus.protocol.application.lighting.LightingSAL], bytes]
Do not call this method directly – use LightingSAL.decode

encode() → bytes
Encodes the SAL into a format for sending over the C-Bus network.

class cbus.protocol.application.lighting.**LightingOnSAL**(*group_address: int*)
Bases: *cbus.protocol.application.lighting.LightingSAL*

Lighting on event SAL

Instructs a given group address to turn it's load on.

This should not be called directly by your code!

Use one of the subclasses of cbus.protocol.lighting.LightingSAL instead.

static decode(*data: bytes*, *command_code: int*, *group_address: int*) → Tuple[cbus.protocol.application.lighting.LightingOnSAL, bytes]
Do not call this method directly – use LightingSAL.decode

encode()
Encodes the SAL into a format for sending over the C-Bus network.

class cbus.protocol.application.lighting.**LightingOffSAL**(*group_address: int*)
Bases: *cbus.protocol.application.lighting.LightingSAL*

Lighting off event SAL

Instructs a given group address to turn it's load off.

This should not be called directly by your code!

Use one of the subclasses of cbus.protocol.lighting.LightingSAL instead.

static decode(*data: bytes*, *command_code: int*, *group_address: int*) → Tuple[cbus.protocol.application.lighting.LightingOffSAL, bytes]
Do not call this method directly – use LightingSAL.decode

encode()
Encodes the SAL into a format for sending over the C-Bus network.

class cbus.protocol.application.lighting.**LightingTerminateRampSAL**(*group_address: int*)
Bases: *cbus.protocol.application.lighting.LightingSAL*

Lighting terminate ramp event SAL

Instructs the given group address to discontinue any ramp operations in progress, and use the brightness that they are currently at.

This should not be called directly by your code!

Use one of the subclasses of cbus.protocol.lighting.LightingSAL instead.

static decode(*data: bytes*, *command_code: int*, *group_address: int*) → Tuple[cbus.protocol.application.lighting.LightingTerminateRampSAL, bytes]
Do not call this method directly – use LightingSAL.decode

encode()
Encodes the SAL into a format for sending over the C-Bus network.

Temperature Broadcast Application

The Temperature Broadcast application is used to notify units on the C-Bus network of changes in temperature. It is used to allow temperature control systems to react to changes in environmental conditions.

This has been replaced by the Measurement application (not get implemented by `libcbus`).

Please refer to this document in conjunction with the Temperature Broadcast Application Guide published by Clipsal.

class cbus.protocol.application.temperature.**TemperatureApplication**
Bases: cbus.protocol.application.sal.BaseApplication

This class is called in the cbus.protocol.applications.APPLICATIONS dict in order to describe how to decode temperature broadcast application events recieved from the network.

Do not call this class directly.

static decode_sals(*data: bytes*) → Sequence[cbus.protocol.application.temperature.TemperatureSAL]
Decodes a temperature broadcast application packet and returns it's SAL(s).

static supported_applications() → Set[cbus.common.Application]
Gets a list of supported Application IDs for the application.

All application IDs must be in the range 0x00 - 0xff.

class cbus.protocol.application.temperature.**TemperatureSAL**(*group_address: int*)
Bases: cbus.protocol.application.sal.SAL, abc.ABC

Base type for temperature broadcast application SALs.

This should not be called directly by your code!

Use one of the subclasses of cbus.protocol.temperature.TemperatureSAL instead.

application

static decode_sals(*data: bytes*) → Sequence[cbus.protocol.application.temperature.TemperatureSAL]
Decodes a temperature broadcast application packet and returns its SAL(s).

Parameters **data** – SAL data to be parsed.

Returns The SAL messages contained within the given data.

Return type list of cbus.protocol.application.temperature.TemperatureSAL

encode() → bytes
Encodes the SAL into a format for sending over the C-Bus network.

class cbus.protocol.application.temperature.**TemperatureBroadcastSAL**(*group_address: int*, *temperature: float*)
Bases: *cbus.protocol.application.temperature.TemperatureSAL*

Temperature broadcast event SAL.

Informs the network of the current temperature being sensed at a location.

Creates a new SAL Temperature Broadcast message.

Parameters

- **group_address** (*int*) – The group address that is reporting the temperature.
- **temperature** (*float*) – The temperature, in degrees celsius, between 0.0 and 63.75.

classmethod decode(*data: bytes*, *group_address: int*) → Tuple[cbus.protocol.application.temperature.TemperatureSAL, bytes]
Do not call this method directly – use TemperatureSAL.decode

encode() → bytes
Encodes the SAL into a format for sending over the C-Bus network.

8.1.3 toolkit Package

cbz Module

cbus/toolkit/cbz.py Library for reading CBus Toolkit CBZ files.

class cbus.toolkit.cbz.**Application**(*oid: uuid.UUID, tag_name: str, address: int, description: str, groups: Sequence[cbus.toolkit.cbz.Group]*)
Bases: *cbus.toolkit.cbz.BaseNetworkElement*

class cbus.toolkit.cbz.**BaseCBZElement**(*oid: uuid.UUID*)
Bases: cbus.toolkit.cbz._Element

class cbus.toolkit.cbz.**BaseNetworkElement**(*oid: uuid.UUID*, *tag_name: str*, *address: int*, *description: str*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**CBZ**(*fh: BinaryIO*)
Bases: object

Opens the file as a CBZ.

exception cbus.toolkit.cbz.**CBZException**
Bases: Exception

class cbus.toolkit.cbz.**Group**(*oid: uuid.UUID, tag_name: str, address: int, description: str, levels: Sequence[cbus.toolkit.cbz.Level]*)
Bases: *cbus.toolkit.cbz.BaseNetworkElement*

class cbus.toolkit.cbz.**Installation**(*oid: uuid.UUID*, *db_version: str*, *version: str*, *modified: datetime.datetime*, *installation_detail: cbus.toolkit.cbz.InstallationDetail*, *project: cbus.toolkit.cbz.Project*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**InstallationDetail**(*oid: uuid.UUID*, *system_location: str*, *hardware_platform: str*, *hostname: str*, *os_name: str*, *os_version: str*, *hardware_location: str*, *installer: cbus.toolkit.cbz.Installer*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**Installer**(*oid: uuid.UUID*, *name: str*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**Interface**(*oid: uuid.UUID*, *interface_type: str*, *interface_address: str*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**Level**(*oid: uuid.UUID*, *tag_name: str*, *address: int*, *description: str*, *value: int*)
Bases: *cbus.toolkit.cbz.BaseNetworkElement*

class cbus.toolkit.cbz.**Network**(*oid: uuid.UUID, tag_name: str, address: int, description: str, network_number: int, interface: cbus.toolkit.cbz.Interface, applications: Sequence[cbus.toolkit.cbz.Application], units: Sequence[cbus.toolkit.cbz.Unit]*)
Bases: *cbus.toolkit.cbz.BaseNetworkElement*

class cbus.toolkit.cbz.**PP**(*name: str*, *value: str*)
Bases: cbus.toolkit.cbz._Element

class cbus.toolkit.cbz.**Project**(*oid: uuid.UUID, tag_name: str, address: str, description: str, network: Sequence[cbus.toolkit.cbz.Network]*)
Bases: *cbus.toolkit.cbz.BaseCBZElement*

class cbus.toolkit.cbz.**Unit**(*oid: uuid.UUID, tag_name: str, address: int, description: str, unit_type: str, unit_name: str, serial_number: str, firmware_version: str, catalog_number: str, pp: Sequence[cbus.toolkit.cbz.PP]*)
Bases: *cbus.toolkit.cbz.BaseNetworkElement*

CHAPTER 9

Indices and tables

- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

S

T

U

V