pyXCP Documentation

Release 0.30.2

Christoph Schueler

Oct 31, 2026

CONTENTS

1 Readme **3**
- 1.1 Installation . . . 3
- 1.2 Quick start . . . 4
- 1.3 Features . . . 5
- 1.4 Documentation . . . 6
- 1.5 Compatibility . . . 6
- 1.6 Contributing . . . 6
- 1.7 License . . . 6
- 1.8 References . . . 6
- 1.9 About ASAM MCD-1 XCP . . . 6
- 1.10 XCP in a nutshell . . . 7
- 1.11 Relation to A2L (ASAM MCD-2 MC) . . . 7
- 1.12 Transports and addressing . . . 7
- 1.13 Supported features (overview) . . . 8
- 1.14 Compliance and versions . . . 8
- 1.15 Safety, performance, and limitations . . . 8
- 1.16 Further resources . . . 8

2 pyXCP Quickstart Guide **9**
- 2.1 Prerequisites . . . 9
- 2.2 Installation . . . 9
- 2.3 Your First XCP Connection . . . 9
- 2.4 Reading and Writing Parameters . . . 10
- 2.5 Basic DAQ Recording . . . 11
- 2.6 Configuration Options . . . 12
- 2.7 Next Steps . . . 12

3 Installation and Getting Started **15**
- 3.1 Installation Methods . . . 15
- 3.2 Requirements . . . 15

4 Platform Setup Guide **17**
- 4.1 Quick Start by Platform . . . 17
- 4.2 Windows Setup . . . 17
- 4.3 Linux Setup . . . 18
- 4.4 macOS Setup . . . 19
- 4.5 Docker and CI/CD . . . 19
- 4.6 Troubleshooting . . . 19

5 pyXCP Tutorial **21**

5.1 Introduction . . . 21
5.2 Installation . . . 21
5.3 Basic Usage . . . 21
5.4 Data Acquisition (DAQ) . . . 24
5.5 Calibration . . . 25
5.6 Advanced Features . . . 26
5.7 Flashing / Programming (overview) . . . 28
5.8 Troubleshooting . . . 29
5.9 Next Steps . . . 29

6 XCP + A2L Integration Guide 31
6.1 Overview . . . 31
6.2 Tools Overview . . . 31
6.3 Prerequisites . . . 32
6.4 Quick Start: Read Parameter by Name . . . 32
6.5 Complete Workflow: Calibration with A2L . . . 33
6.6 DAQ Setup with A2L Metadata . . . 34
6.7 A2L File Structure . . . 35
6.8 Data Type Conversion . . . 36
6.9 Working Example . . . 36
6.10 Advanced: Production Workflows with asamint . . . 37
6.11 Troubleshooting . . . 38
6.12 FAQ . . . 38
6.13 Related Examples . . . 39
6.14 References . . . 39

7 Configuration 41
7.1 Configuration Systems . . . 41
7.2 Traitlets-based Configuration . . . 41
7.3 Legacy Configuration (Deprecated) . . . 41
7.4 Transport Layer Configuration . . . 42
7.5 Additional Notes . . . 43

8 Logging Configuration 45
8.1 Overview . . . 45
8.2 Default Behavior . . . 45
8.3 Logger Hierarchy . . . 45
8.4 Enabling Logging . . . 46
8.5 Advanced Configuration . . . 47
8.6 Logging Levels . . . 49
8.7 Troubleshooting . . . 49
8.8 Examples . . . 50
8.9 FAQ . . . 51
8.10 Related Documentation . . . 52
8.11 References . . . 52

9 Frame Acquisition Policy Migration Guide 53
9.1 Overview . . . 53
9.2 Why the Change? . . . 53
9.3 Available Policies . . . 53
9.4 Migration Checklist . . . 55
9.5 Performance Comparison . . . 56
9.6 Examples . . . 56
9.7 Related Issues . . . 56
9.8 References . . . 57

10 Custom CAN bus objects **59**
10.1 python-can plugin interface (recommended for third-party drivers) . . . 59
10.2 Using an existing CAN bus object . . . 60
10.3 Fully custom interface (hardware not supported by python-can) . . . 61

11 HOW-TOs **63**

12 How-to build your own CAN drivers **65**
12.1 Using an existing CAN bus object . . . 65
12.2 Using an unsupported interface . . . 65
12.3 Important Notes . . . 66

13 Using pyXCP Command-Line Tools **67**
13.1 Quick Reference . . . 67
13.2 Available Command-Line Tools . . . 67
13.3 Basic Usage . . . 68
13.4 Examples . . . 68
13.5 Using the xcp-profile Tool . . . 68

14 Command-Line Tools Reference **71**
14.1 Quick Reference . . . 71
14.2 Configuration Methods . . . 71
14.3 Tool Details . . . 72

15 Recorder **75**
15.1 Overview . . . 75
15.2 Quick start example (run_daq) . . . 75
15.3 xcp_daq_recorder Scipt . . . 76
15.4 Online vs offline and Policies . . . 77
15.5 Timestamps . . . 77
15.6 Setting up DAQ lists . . . 77
15.7 Post-processing .xmraw recordings . . . 78
15.8 Converters and examples . . . 79
15.9 Miscellaneous notes . . . 79

16 pyxcp package **81**
16.1 Subpackages . . . 81
16.2 Submodules . . . 82
16.3 pyxcp.checksum module . . . 82
16.4 pyxcp.cmdline module . . . 82
16.5 pyxcp.constants module . . . 82
16.6 pyxcp.dllif module . . . 82
16.7 pyxcp.errormatrix module . . . 82
16.8 pyxcp.time_correlation module . . . 82
16.9 pyxcp.time_sync module . . . 82
16.10 pyxcp.timing module . . . 82
16.11 pyxcp.types module . . . 82
16.12 pyxcp.utils module . . . 82
16.13 Module contents . . . 82

17 Indices and tables **83**

CHAPTER

ONE

README

Reliable Python tooling for the ASAM MCD-1 XCP protocol (measurement, calibration, flashing) with multiple transports (CAN, Ethernet, USB, Serial) and handy CLI utilities.

`pyXCP` is a production-ready Python library for communicating with XCP-enabled devices, most commonly automotive ECUs. Use it to take measurements, adjust parameters (calibration), stream DAQ/STIM, and program devices during development.

Highlights:

- Transports: Ethernet (TCP/IP), CAN, USB, Serial (SxI)
- Cross-platform: Windows, Linux, macOS
- Rich CLI tools for common XCP tasks
- Extensible architecture and layered design

1.1 Installation

The easiest way is from PyPI:

```
pip install pyxcp
```

To install from the main branch:

```
pip install git+https://github.com/christoph2/pyxcp.git
```

1.1.1 Requirements

- Python >= 3.11, < 3.16
- Building from source requires a working C/C++ toolchain (native extensions are used for performance). Wheels are provided for common platforms and Python versions; if a wheel is not available, pip will build from source.
- An XCP slave device (or simulator)

1.2 Quick start

New to pyXCP? Start with the 15-minute Quickstart Guide for a hands-on introduction.

The comprehensive tutorial walks you through typical tasks end-to-end.

Using A2L files? See the A2L Integration Guide for symbolic access to ECU parameters.

Minimal example using the built-in argument parser and context manager:

```
from pyxcp.cmdline import ArgumentParser

ap = ArgumentParser(description="pyXCP hello world")

with ap.run() as x:
    x.connect()
    identifier = x.identifier(0x01)
    print(f"ID: {identifier!r}")
    print(x.slaveProperties)
    x.disconnect()
```

Async facade for awaitable command handling plus DAQ/event streaming:

```
import asyncio

from pyxcp.cmdline import ArgumentParser
from pyxcp.transport import AsyncPolicyAdapter


async def main():
    ap = ArgumentParser(description="pyXCP async example")
    policy = AsyncPolicyAdapter()

    async with ap.async_run(policy=policy) as x:
        await x.connect()
        status = await x.getStatus()
        print(status)

        daq_frames = x.subscribe_daq()
        frame = await daq_frames.get()
        print(frame.category, frame.counter, frame.payload.hex())


asyncio.run(main())
```

1.2.1 Configuration

pyXCP supports a traitlets-based configuration system.

- Recommended Python config example and generator: tutorial and configuration
- Legacy TOML examples remain available for compatibility.

1.2.2 Command-line tools

Installed entry points (see pyproject.toml):

- xcp-info — print capabilities and properties
- xcp-id-scanner — scan for slave identifiers
- xcp-fetch-a2l — retrieve A2L from target (if supported)
- xcp-profile — generate/convert config files
- xcp-examples — launch assorted demos/examples
- xcp-discovery — multicast discovery for XCP on Ethernet (with optional SET_SLAVE_IP_ADDRESS)
- xmraw-converter — convert recorder .xmraw data
- pyxcp-probe-can-drivers — list available CAN interfaces

Run any tool with -h for options.

1.3 Features

- Multiple transport layers: Ethernet (TCP), CAN, USB, SxI (serial/UART)
- High-precision IEEE 1588/PTP hardware timestamping (Ethernet/UDP on Windows and Linux)
- Data Acquisition (DAQ) and Stimulation (STIM)
- Calibration (read/write parameters)
- Flashing/programming workflows
- **A2L (ASAM MCD-2 MC) support** — symbolic access via pya2ldb
- Recorder utilities and converters (see recorder)
- Extensible architecture for custom transports

1.3.1 Related Projects

asamint — High-level MCS (Measurement & Calibration System)

For production-grade workflows with command-line batch operations, see the asamint project.

`asamint` integrates `pyxcp`, `pya2ldb`, `asammdf`, and `objutils` to provide:

- Command-line MCS functionality
- ASAM CDF (calibration data file) creation
- MDF (ASAM MCD-3 MC) export
- Orchestrated multi-tool workflows
- Production measurement campaigns

When to use:

- `pyxcp`: Custom applications, test automation, learning XCP
- `asamint`: Production calibration, batch operations, command-line MCS

1.4 Documentation

- Getting started tutorial: tutorial
- **A2L Integration Guide**: a2l_integration
- Configuration: configuration
- CAN driver setup and troubleshooting: howto_can_driver
- Recorder: recorder
- Troubleshooting: troubleshooting
- Troubleshooting matrix (common errors, root causes, fixes): troubleshooting_matrix

To build the Sphinx documentation locally:

1. **Install doc requirements:**
   ```
   pip install -r docs/requirements.txt
   ```
2. **Build:**
   ```
   sphinx-build -b html docs docs/_build/html
   ```
3. **Open**
   ```
   docs/_build/html/index.html
   ```

1.5 Compatibility

- Operating systems: Windows, Linux, macOS
- Python: 3.11 - 3.15, CPython wheels where available
- CAN backends: python-can compatible drivers (see howto_can_driver)

1.6 Contributing

Contributions are welcome! Please: - Read CODE_OF_CONDUCT - Open an issue or discussion before large changes - Use pre-commit to run linters and tests locally

1.7 License

GNU Lesser General Public License v3 or later (LGPLv3+). See LICENSE for details.

1.8 References

- ASAM MCD-1 XCP standard: https://www.asam.net/standards/detail/mcd-1-xcp/

1.9 About ASAM MCD-1 XCP

XCP (Universal Measurement and Calibration Protocol) is an ASAM standard defining a vendor-neutral protocol to access internal data of electronic control units (ECUs) for measurement, calibration (parameter tuning), and programming. XCP decouples the protocol from the physical transport, so the same command set can be carried over different buses such as CAN, FlexRay, Ethernet, USB, or Serial.

- Roles: An XCP Master (this library) communicates with an XCP Slave (your device/ECU or simulator).

- Layered concept: XCP defines an application layer and transport layers. pyXCP implements the application layer and multiple transport bindings.
- Use cases:
  - Measurement: Read variables from the ECU in real-time, including high-rate DAQ streaming.
  - Calibration: Read/write parameters (calibration data) in RAM/flash.
  - Programming: Download new program/data to flash (where the slave supports it).

For the authoritative description, see the ASAM page: https://www.asam.net/standards/detail/mcd-1-xcp/

1.10 XCP in a nutshell

- Connect/Session: The master establishes a connection, negotiates capabilities/features, and optionally unlocks protected functions via seed & key.
- Addressing: Memory is accessed via absolute or segment-relative addresses. Addressing modes are described in the associated A2L file (ASAM MCD-2 MC), which maps symbolic names to addresses, data types, and conversion rules.
- Events: The slave exposes events (e.g., "1 ms task", "Combustion cycle"), which trigger DAQ sampling. The master assigns signals (ODTs) to these events for time-aligned acquisition.
- DAQ/STIM: DAQ = Data Acquisition (slave → master), STIM = Stimulation (master → slave). Both use event-driven lists for deterministic timing.
- Timestamps: DAQ may carry timestamps from the slave for precise time correlation.
- Security: Access to sensitive commands (e.g., programming, calibration) can be protected by a seed & key algorithm negotiated at runtime.
- Checksums: XCP defines checksum services useful for verifying memory regions (e.g., after flashing).

1.11 Relation to A2L (ASAM MCD-2 MC)

While XCP defines the protocol, the A2L file describes the measurement and calibration objects (characteristics, measurements), data types, conversion rules, and memory layout. In practice, you use pyXCP together with an A2L to: - Resolve symbolic names to addresses and data types. - Configure DAQ lists from human-readable signal names. - Interpret raw values using the appropriate conversion methods.

pyXCP provides utilities to fetch A2L data when supported by the slave and to work with A2L-described objects. See also pya2ldb!

1.12 Transports and addressing

XCP is transport-agnostic. pyXCP supports multiple transports and addressing schemes: - CAN (XCP on CAN): Robust and ubiquitous in vehicles; limited payload and bandwidth; suited for many calibration tasks and moderate DAQ rates. - Ethernet (XCP on TCP/UDP): High bandwidth with low latency; well suited for rich DAQ and programming workflows. - USB: High throughput for lab setups; requires device support. - Serial/SxI: Simple point-to-point links for embedded targets and simulators.

The exact capabilities (e.g., max CTO/DTO, checksum types, timestamping) are negotiated at connect time and depend on the slave and transport.

1.13 Supported features (overview)

The scope of features depends on the connected slave. At the library level, pyXCP provides: - Session management: CONNECT/DISCONNECT, GET_STATUS/SLAVE_PROPERTIES, communication mode setup, error handling. - Memory access: Upload/short upload, Download/Download Next, verifications, optional paged memory where supported. - DAQ/STIM: Configuration of DAQ lists/ODTs, event assignment, data streaming, timestamp handling when available. - Programming helpers: Building blocks for program/erase/write flows (exact sequence per slave's flash algorithm and A2L description). - Security/Seed & Key: Pluggable seed-to-key resolution including 3264-bit bridge on Windows. - Utilities: Identifier scanning, A2L helpers, recorder and converters.

Refer to tutorial and configuration for feature usage, and xcp-info for a capability dump of your target.

1.14 Compliance and versions

pyXCP aims to be compatible with commonly used parts of ASAM MCD-1 XCP. Specific optional features are enabled when a slave advertises them during CONNECT. Because implementations vary across vendors and ECU projects, always consult your A2L and use xcp-info to confirm negotiated options (e.g., checksum type, timestamp unit, max DTO size, address granularity).

If you rely on a particular XCP feature/profile not mentioned here, please open an issue with details about your slave and A2L so we can clarify support and—if feasible—add coverage.

1.15 Safety, performance, and limitations

- Safety-critical systems: XCP is a development and testing protocol. Do not enable measurement/calibration on safety-critical systems in the field unless your system-level safety case covers it.
- Performance: Achievable DAQ rates depend on transport bandwidth, ECU event rates, DTO sizes, and host processing. Ethernet typically yields the highest throughput.
- Latency/jitter: Event scheduling in the slave and OS scheduling on the host can affect determinism. Use timestamps to correlate data precisely.
- Access control: Seed & key protects sensitive functions; your organization's policy should govern algorithm distribution and access.

1.16 Further resources

- ASAM MCD-1 XCP standard (overview and membership): https://www.asam.net/standards/detail/mcd-1-xcp/
- ASAM MCD-2 MC (A2L) for object descriptions: https://www.asam.net/standards/detail/mcd-2-mc/
- Introduction to DAQ/STIM concepts (ASAM publications and vendor docs)
- Related: CCP (legacy predecessor to XCP), ASAM MDF for measurement data storage

CHAPTER

TWO

PYXCP QUICKSTART GUIDE

Get started with pyXCP in 15 minutes: installation, first connection, parameter read/write, and basic DAQ recording.

2.1 Prerequisites

- Python 3.8+ (64-bit recommended)
- XCP slave device (ECU, simulator, or test tool)
- Transport interface: CAN adapter, Ethernet, USB, or serial
- Optional: A2L file (ASAM MCD-2 MC) for symbolic access

Safety: XCP is for development/testing—avoid safety-critical systems without proper analysis.

2.2 Installation

Basic installation:

```
pip install pyxcp
python -c "import pyxcp; print(pyxcp.__version__)"
```

Optional dependencies:

```
# A2L support
pip install pyxcp pya2ldb

# CAN drivers
pip install pyxcp python-can[pcan]
pip install pyxcp python-can[vector]
pip install pyxcp python-can[ixxat]
```

2.3 Your First XCP Connection

Minimal CAN example:

```
from pyxcp import Master

with Master("can") as xcp:
    xcp.connect()
    ecu_id = xcp.getId(0x01)
```

(continues on next page)

(continued from previous page)

```
    print(f"Connected to ECU: {ecu_id}")
    xcp.disconnect()
```

Run with CLI args:

```
python your_script.py --transport CAN --device socketcan --channel can0 --bitrate 500000
```

Ethernet example (TCP):

```
from pyxcp import Master

with Master("eth") as xcp:
    xcp.connect()
    props = xcp.slaveProperties
    print(f"Protocol Layer: {props.protocolLayerVersion}")
    print(f"Max CTO/DTO: {props.maxCto}/{props.maxDto}")
    xcp.disconnect()
```

No config file required (programmatic):

```
from pyxcp import Master
from pyxcp.config import create_application_from_config, set_application

config = {"Transport": {"CAN": {"device": "socketcan", "channel": "can0", "bitrate":␣
↪500000, "max_dlc": 8}}}
app = create_application_from_config(config)
set_application(app)

with Master("can") as xcp:
    xcp.connect()
    print(f"ECU ID: {xcp.getId(0x01)}")
    xcp.disconnect()
```

2.4 Reading and Writing Parameters

Upload (read):

```
from pyxcp import Master
import struct

with Master("can") as xcp:
    xcp.connect()
    data = xcp.upload(address=0x1A2000, length=4)
    value = struct.unpack("<I", data)[0]
    print(f"Value: {value}")
    xcp.disconnect()
```

Download (write):

```
from pyxcp import Master
import struct
```

(continues on next page)

(continued from previous page)

```
with Master("can") as xcp:
    xcp.connect()
    new_value = 42
    xcp.download(address=0x1A2000, data=struct.pack("<I", new_value))
    readback = struct.unpack("<I", xcp.upload(address=0x1A2000, length=4))[0]
    print(f"Written: {new_value}, Read back: {readback}")
    xcp.disconnect()
```

2.5 Basic DAQ Recording

Heads-up (Issue #253): Some slaves lack optional DAQ services. `getDaqInfo()` returns `valid` flags (`processor`, `resolution`, `events`). If processor/resolution are `False`, supply trusted DAQ info via `DaqProcessor.setup(daq_info_override=...)` or abort.

Simple DAQ example:

```
from pyxcp import Master
from pyxcp.daq_stim import DaqList, DaqToCsv
import time

with Master("can") as xcp:
    xcp.connect()
    daq_info = xcp.getDaqInfo()
    if not daq_info["valid"]["processor"] or not daq_info["valid"]["resolution"]:
        raise RuntimeError("Slave did not provide DAQ capabilities; supply overrides␣
↪before proceeding.")

    daq_list = DaqList(
        name="Engine",
        event=0,
        measurements=[
            {"address": 0x1A2000, "ext": 0, "size": 4},
            {"address": 0x1A2004, "ext": 0, "size": 4},
            {"address": 0x1A2008, "ext": 0, "size": 2},
        ],
    )

    policy = DaqToCsv([daq_list])
    xcp.setupDaq([daq_list], policy)
    xcp.startDaq()
    time.sleep(10)
    xcp.stopDaq()
    xcp.disconnect()
```

DAQ with data conversion:

```
import struct
from pyxcp.daq_stim import DaqList

def convert_engine_speed(raw_bytes):
    return struct.unpack("<I", raw_bytes)[0] * 0.25
```

(continues on next page)

(continued from previous page)

```
def convert_temperature(raw_bytes):
    return (struct.unpack("<I", raw_bytes)[0] * 0.1) - 40.0

measurements = [
    {"address": 0x1A2000, "ext": 0, "size": 4, "name": "EngineSpeed_RPM", "conversion":␣
↪convert_engine_speed},
    {"address": 0x1A2004, "ext": 0, "size": 4, "name": "EngineTemp_C", "conversion":␣
↪convert_temperature},
]
```

2.6 Configuration Options

Method 1: Command-line (recommended):

```
from pyxcp.cmdline import ArgumentParser

ap = ArgumentParser(description="My XCP Tool")
with ap.run() as xcp:
    xcp.connect()
    # ...
    xcp.disconnect()
```

Run with:

```
python tool.py --transport CAN --device socketcan --channel can0 --bitrate 500000
```

Method 2: Config file (`pyxcp_conf.py`):

```
c = get_config()
c.Transport.CAN.device = "socketcan"
c.Transport.CAN.channel = "can0"
c.Transport.CAN.bitrate = 500000
```

Config search order: `PYXCP_CONFIG` env var → CWD → script dir → `~/.pyxcp/pyxcp_conf.py`.

Method 3: Programmatic (embedding):

```
from pyxcp.config import create_application_from_config, set_application
config = {"Transport": {"CAN": {"device": "socketcan", "channel": "can0", "bitrate":␣
↪500000}}}
app = create_application_from_config(config)
set_application(app)
```

2.7 Next Steps

- *pyXCP Tutorial* – comprehensive guide with advanced topics
- FAQ – common questions and solutions
- Examples: `pyxcp/examples` directory
- API Reference: `pyxcp.rst`

Example usage:

```
python xcphello.py --transport CAN --device socketcan --channel can0
python run_daq.py --transport CAN --device socketcan --channel can0
```

CHAPTER

THREE

INSTALLATION AND GETTING STARTED

Python: *Python* >= 3.11, < 3.16

Platforms: No platform-specific restrictions besides availability of communication drivers (CAN, Ethernet, USB, etc.).

Documentation: Latest documentation

3.1 Installation Methods

3.1.1 From PyPI

```
pip install pyxcp
```

3.1.2 From Source

```
# Clone the repository
git clone https://github.com/christoph2/pyxcp.git
cd pyxcp

# Install dependencies
pip install -r requirements.txt

# Install the package
python setup.py install
```

3.1.3 Using pip with GitHub

```
pip install git+https://github.com/christoph2/pyxcp.git
```

3.2 Requirements

- Python >= 3.11, < 3.16
- A running XCP slave (of course)
- If you are using a 64-bit Windows version and want to use seed-and-key .dlls (to unlock resources), a GCC compiler capable of creating 32-bit executables is required. These .dlls almost always ship as 32-bit versions, but you can't load a 32-bit .dll into a 64-bit process, so a small bridging program (asamkeydll.exe) is required.

CHAPTER

FOUR

PLATFORM SETUP GUIDE

pyXCP runs on Windows, Linux, and macOS. This guide covers platform-specific installation, C++ extension compilation, CAN driver setup, and common issues.

4.1 Quick Start by Platform

Windows:

```
# Install Python 3.11 through 3.15
pip install pyxcp
# For CAN: install vendor driver
# For source builds: install Visual Studio Build Tools
pip install pyxcp --no-binary pyxcp  # to force build
```

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install python3-pip python3-dev build-essential cmake
pip install pyxcp
sudo apt install can-utils
sudo ip link set can0 up type can bitrate 500000
```

macOS:

```
xcode-select --install
brew install python@3.12
pip3 install pyxcp
# CAN options are limited; see CAN section
```

4.2 Windows Setup

Python installation: use Python 3.11 through 3.15 from python.org; add to PATH; enable pip; consider disabling path length limit.

Verify:

```
python --version
pip --version
```

Install pyXCP (binary wheel recommended):

```
pip install pyxcp
```

Build from source:

```
pip install pyxcp --no-binary pyxcp
```

C++ build tools:

1. Visual Studio Build Tools (workload "Desktop development with C++")
2. Includes MSVC, CMake, Windows SDK

Verify:

```
where cl
cmake --version
```

4.2.1 CAN drivers on Windows

pyXCP uses python-can backends.

- Vector: install Vector drivers, then `pip install python-can[vector]`; verify with `pyxcp-probe-can-drivers`.
- PEAK PCAN: install driver + PCAN-View; `pip install python-can[pcan]`.
- Kvaser: install CANlib SDK; `pip install python-can[kvaser]`.
- IXXAT: install VCI; `pip install python-can[ixxat]`.
- Virtual CAN: use python-can virtual backend.

Example config factory:

```
def create_transport(parent):
    from pyxcp.transport.can import Can
    return Can(parent, can_interface="vector", can_channel=0,
               can_id_master=0x700, can_id_slave=0x701, can_bitrate=500000)
```

4.2.2 Seed/Key DLLs (32→64-bit bridge)

Problem: OEM seed/key DLLs are often 32-bit while Python is 64-bit.

Solution: pyXCP includes `asamkeydll.exe` bridge. Ensure MinGW-w64 is installed if rebuilding is needed; configure `SEED_KEY_DLL = "SeedNKeyXcp.dll"`.

4.3 Linux Setup

Dependencies:

```
sudo apt update
sudo apt install python3-pip python3-dev build-essential cmake libpython3-dev
pip install pyxcp
```

SocketCAN example:

```
sudo ip link set can0 up type can bitrate 500000
pyxcp-probe-can-drivers
```

4.4 macOS Setup

Install Xcode Command Line Tools and Homebrew Python. Build tools (CMake) come with Homebrew. CAN support depends on vendor hardware; consult vendor docs.

4.5 Docker and CI/CD

- Use manylinux wheels on Linux where possible.
- For source builds in CI: install build-essential, cmake, python3-dev (Linux) or MSVC tools (Windows).

4.6 Troubleshooting

- Missing compiler: install platform build tools (MSVC, Xcode CLT, GCC/CMake).
- CAN not working: verify driver installation and interface name; use `pyxcp-probe-can-drivers`.
- Firewall (Ethernet): allow Python on required ports or test on trusted network.

CHAPTER

FIVE

PYXCP TUTORIAL

This tutorial will guide you through the basics of using the pyXCP library to communicate with XCP-enabled devices.

5.1 Introduction

pyXCP is a Python library for communicating with devices that support the XCP (Universal Measurement and Calibration Protocol) protocol. XCP is commonly used in automotive applications for calibration, measurement, and flashing of ECUs (Electronic Control Units).

5.2 Installation

Install pyXCP from PyPI:

```
pip install pyxcp
```

For building from source and development details, see the project README.

5.3 Basic Usage

5.3.1 Before you start

- Prerequisites: An XCP slave (ECU, device, or simulator), and ideally its A2L file (ASAM MCD-2 MC). The A2L maps symbolic names to addresses, data types, and conversions used for calibration/measurement. Without an A2L you can still work with raw addresses.
- Safety: XCP is for development and testing. Avoid enabling measurement/calibration on safety-critical systems unless your safety case covers it. Prefer a lab setup for first steps.
- Transport choice: pyXCP supports Ethernet, CAN, USB, and Serial (SxI). Ethernet usually offers the highest throughput; CAN is ubiquitous and robust but has limited payload. See also *How-to build your own CAN drivers* for CAN setup.

5.3.2 Choose your transport (quick guide)

- Ethernet (TCP): If your device exposes XCP on TCP, you'll typically specify host/port.
- CAN: Requires a python-can compatible driver and correct CAN IDs/filters. Use pyxcp-probe-can-drivers to list available interfaces. See *How-to build your own CAN drivers*.
- USB/Serial: Device-specific; consult your device documentation.

You can configure transports either by command line via the built-in ArgumentParser or via the traitlets configuration system (see Configuration below).

5.3.3 CLI quick wins

Try these before writing code:

```
# Discover basic capabilities and negotiated options
xcp-info -t eth --host 127.0.0.1 --port 5555

# Scan for identifier packets (e.g., on CAN)
xcp-id-scanner -t can --driver kvaser --channel 0 --bitrate 500000

# Attempt to fetch A2L from the target if supported
xcp-fetch-a2l -t eth --host 127.0.0.1 --port 5555 -o my_ecu.a2l

# See available demos
xcp-examples -h
```

5.3.4 Connecting to an XCP Slave

The most basic operation is to connect to an XCP slave device and retrieve information about it:

```
from pyxcp.cmdline import ArgumentParser

# Create an argument parser that handles common XCP connection parameters
ap = ArgumentParser(description="pyXCP hello world example")

# Use a context manager to ensure proper cleanup
with ap.run() as x:
    # Connect to the XCP slave
    x.connect()

    # Get the slave identifier
    identifier = x.identifier(0x01)
    print(f"ID: {identifier!r}")

    # Print slave properties
    print(x.slaveProperties)

    # Disconnect when done
    x.disconnect()
```

The `ArgumentParser` class handles command-line arguments for specifying the transport layer (CAN, Ethernet, USB, etc.) and connection parameters. It can also wrap an existing `argparse.ArgumentParser` to support custom application-specific arguments:

```
import argparse
from pyxcp.cmdline import ArgumentParser

# 1. Create a standard argparse parser for your custom options
parser = argparse.ArgumentParser(description="My custom tool")
parser.add_argument("--my-option", action="store_true", help="Custom flag")

# 2. Wrap it with pyXCP's ArgumentParser
ap = ArgumentParser(parser)
```

(continues on next page)

(continued from previous page)

```
# 3. Access both pyXCP and custom arguments
with ap.run() as x:
    if ap.args.my_option:
        print("Custom option enabled!")
    # ... use the master instance x ...
```

Examples: Ethernet and CAN invocations

- Ethernet (TCP):

```
python your_script.py -t eth --host 127.0.0.1 --port 5555 --protocol TCP
```

- CAN (Kvaser example):

```
python your_script.py -t can --driver kvaser --channel 0 --bitrate 500000 \
  --can-id-master 0x300 --can-id-slave 0x301
```

Tip: If you prefer config files, you can omit many CLI flags and specify them in a traitlets config (see Configuration below). Use `xcp-profile create -o my_config.py` to generate a template.

5.3.5 Configuration

pyXCP supports two configuration systems:

Traitlets-based Configuration (Recommended)

The recommended way to configure pyXCP is using the traitlets-based configuration system with Python configuration files:

```
# Connect using a Python configuration file
python your_script.py -t eth --config path/to/config.py
```

Example Python configuration file (config.py):

```
# Configuration file for pyXCP
c = get_config()  # noqa

# Transport configuration
c.Transport.layer = "ETH"

# Ethernet configuration
c.Transport.Eth.host = "localhost"
c.Transport.Eth.port = 5555
c.Transport.Eth.protocol = "TCP"
```

You can generate a template configuration file with all available options:

```
xcp-profile create -o my_config.py
```

Legacy TOML Configuration

The older TOML-based configuration is still supported but is now considered legacy:

```
# Connect using legacy TOML configuration
python your_script.py -t eth --config path/to/config.toml
```

Example legacy configuration file for Ethernet (conf_eth.toml):

```
[XCP]
TRANSPORT = "ETH"

[ETH]
HOST = "localhost"
PORT = 5555
PROTOCOL = "TCP"
```

Example legacy configuration file for CAN (conf_can.toml):

```
[XCP]
TRANSPORT = "CAN"

[CAN]
CHANNEL = 0
BITRATE = 500000
```

You can convert a legacy configuration file to the new format:

```
xcp-profile convert -c old_config.toml -o new_config.py
```

5.4 Data Acquisition (DAQ)

XCP supports data acquisition for collecting measurement data from the slave device.

Concepts recap: The slave exposes events (e.g., “1 ms task”) that drive sampling. You configure one or more DAQ lists and ODTs, assign them to events, and optionally enable timestamps for precise correlation. DTO size and bandwidth depend on the negotiated transport parameters.

Note: For a higher-level DAQ workflow using Policies (online CSV and offline .xmraw recording) and post-processing, see the Recorder page: *Recorder*. The snippet below demonstrates a low-level DAQ setup directly with the master API:

```
from pyxcp.cmdline import ArgumentParser
from pyxcp.recorder import XcpLogFileWriter

ap = ArgumentParser(description="pyXCP DAQ example")

with ap.run() as x:
    x.connect()

    # Configure DAQ
    x.cond_unlock()

    # Get DAQ information
    daq_info = x.getDaqInfo()
```

(continues on next page)

(continued from previous page)

```
    # Set up a DAQ list
    x.setDaqPtr(0, 0, 0)
    x.writeDaq(0x1234)  # Address to measure

    # Start DAQ
    x.startStopDaqList(0, 1)
    x.startStopSynch(1)

    # Create a recorder to save data
    recorder = XcpLogFileWriter("measurement.xmraw")
    x.set_recorder(recorder)

    # Wait for data
    import time
    time.sleep(5)

    # Stop DAQ
    x.startStopSynch(0)
    recorder.close()

    x.disconnect()
```

5.5 Calibration

XCP allows you to read and write parameters in the slave device:

```
from pyxcp.cmdline import ArgumentParser

ap = ArgumentParser(description="pyXCP calibration example")

with ap.run() as x:
    x.connect()

    # Unlock the slave for calibration
    x.cond_unlock()

    # Read a value
    address = 0x1234
    size = 4  # bytes
    data = x.upload(size, address)
    print(f"Value at 0x{address:X}: {int.from_bytes(data, byteorder='little')}")

    # Write a value
    new_value = 42
    x.download(address, new_value.to_bytes(size, byteorder='little'))

    x.disconnect()
```

5.6 Advanced Features

5.6.1 Handling optional DAQ information (Issue #253)

Some ECUs do not implement all optional DAQ services. Starting with v0.26.x, `getDaqInfo()` returns a `valid` map (`processor`, `resolution`, `events`) to signal whether values came from the slave or safe defaults. If `processor` or `resolution` are `False`, provide trusted data when initializing DAQ:

```
daq_info = x.getDaqInfo(include_event_lists=False)
if not daq_info["valid"]["processor"] or not daq_info["valid"]["resolution"]:
    raise RuntimeError("Slave DAQ info incomplete (see Issue #253); provide overrides.")

# Optional override when the slave is incomplete
daq_override = {...}  # processor/resolution dict from A2L or manual config
daq_processor.setup(daq_info_override=daq_override)
```

If `events` is `False`, the event list is empty (e.g., `GET_DAQ_EVENT_INFO` not implemented); provide event IDs from configuration or skip event-dependent features.

Example override

Use trusted data from A2L or ECU documentation when the slave omits processor/resolution info:

```
daq_override = {
    "processor": {
        "minDaq": 0,
        "maxDaq": 4,
        "properties": {
            "configType": "DYNAMIC",
            "overloadEvent": False,
            "overloadMsb": True,
            "prescalerSupported": False,
            "pidOffSupported": False,
            "timestampSupported": True,
            "bitStimSupported": False,
            "resumeSupported": False,
        },
        "keyByte": {
            "identificationField": "IDF_ABS_ODT_NUMBER",
            "addressExtension": "AE_SAME_FOR_ALL",
            "optimisationType": "OM_DEFAULT",
        },
    },
    "resolution": {
        "timestampTicks": 1,
        "maxOdtEntrySizeDaq": 248,
        "maxOdtEntrySizeStim": 248,
        "granularityOdtEntrySizeDaq": 1,
        "granularityOdtEntrySizeStim": 1,
        "timestampMode": {"unit": "DAQ_TIMESTAMP_UNIT_1NS", "fixed": True, "size": "S4"},
    },
    "channels": [],  # supply if you have event info; leave empty if events are unknown
    "valid": {"processor": True, "resolution": True, "events": False},
}
```

5.6.2 Using Custom Transport Layers

pyXCP supports various transport layers, and you can also create custom ones:

```
from pyxcp.cmdline import ArgumentParser
from pyxcp.transport.can import CanInterfaceBase

# Create a custom CAN interface
class MyCanInterface(CanInterfaceBase):
    def __init__(self, config):
        super().__init__(config)
        # Initialize your custom CAN hardware

    def transmit(self, payload):
        # Implement sending data
        pass

    def receive(self, timeout=None):
        # Implement receiving data
        pass

    def close(self):
        # Clean up resources
        pass

# Register your custom interface
from pyxcp.transport.can import register_can_interface
register_can_interface("my_can", MyCanInterface)

# Now you can use it in your configuration
# [CAN]
# DRIVER = "my_can"
```

5.6.3 Upgrading an Existing Configuration

You can convert a legacy configuration file (TOML/JSON) to the new Python/traitlets format using the xcp-profile tool:

```
xcp-profile convert -c old_config.toml -o new_config.py
```

Note on CAN identifier parameters (issue #130): The new configuration system interprets CAN IDs via

```
c.Transport.Can.can_id_master = 47
c.Transport.Can.can_id_slave = 11
```

correctly. For compatibility, existing legacy files continue to behave as before; when you convert a legacy file, ID roles are normalized in the generated Python config. The logger illustrates the resolved IDs at runtime, for example:

```
2024-08-06 16:25:54 INFO     XCPonCAN - Interface-Type: 'kvaser' Parameters: [('channel',
↪ '0'), ('fd', False), ('bitrate', 500000),
                             ('receive_own_messages', False), ('sjw', 2), ('tseg1', 5), (
↪'tseg2', 2)]
                     INFO     XCPonCAN - Master-ID (Tx): 0x00000300S -- Slave-ID (Rx):␣
↪0x00000301S
2024-08-06 16:25:55 INFO     XCPonCAN - Filters used: [{'can_id': 769, 'can_mask': 2047,
```

(continues on next page)

(continued from previous page)

```
↪'extended': False}]
                    INFO     XCPonCAN - State: BusState.ACTIVE
```

5.6.4 Unlocking via Seed & Key (Python)

Instead of a DLL, you can provide a Python function for Seed & Key handling:

```
def SeedKeyXCP(resource: int, seed: bytes) -> bytes:
    temp0, temp1, temp2, temp3 = seed[0], seed[1], seed[2], seed[3]
    temp = (temp3 << 24) | (temp2 << 16) | (temp1 << 8) | temp0
    temp = (temp >> 5) | (temp << 23)
    temp = (temp * 7) ^ 0x26031961

    key = bytearray(9)
    key[0] = (temp >> 0) & 0xFF
    key[1] = (temp >> 8) & 0xFF
    key[2] = (temp >> 16) & 0xFF
    key[3] = (temp >> 24) & 0xFF
    return bytes(key)

# c.General.seed_n_key_dll = 'SeedNKeyXcp.dll'  # alternative
c.General.seed_n_key_function = SeedKeyXCP
```

5.6.5 Re-using an existing CAN interface

If you already manage a CAN bus object externally, you can pass it into pyXCP. See HOW-TO: How to build your own CAN drivers (howto_can_driver.rst) for a full example and caveats.

5.6.6 Timestamping note

Timestamps are generated by a C++ extension. The application start timestamp (including timezone and offsets) is available on the context:

```
with ap.run() as x:
    print("Start DT:", x.start_datetime)
```

5.7 Flashing / Programming (overview)

Flashing support depends on the slave's programming algorithm and the A2L description. pyXCP provides the building blocks (CONNECT, PROGRAM_RESET, ERASE, DOWNLOAD, VERIFY, checksums), but the exact sequence and memory layout are slave-specific.

- Always back up calibration data where applicable.
- Verify negotiated options (e.g., checksum type, max DTO) with `xcp-info` before large transfers.
- Consult your A2L for programming sections and address granularity.
- For safety, test on a simulator or development device first.

5.8 Troubleshooting

- Timeouts on connect: Check transport parameters (host/port for ETH; channel/bitrate and filters for CAN). Use `xcp-info` or `xcp-id-scanner` to validate connectivity.
- Seed & Key fails: Confirm you provided the correct DLL or Python function. On Windows with 32-bit only DLLs, use the provided 3264

bridge (asamkeydll.exe). See README and *Configuration*. - Wrong or swapped CAN IDs: Ensure `can_id_master` and

`can_id_slave` are set correctly (see Configuration). The logger prints resolved IDs.

- DTO too large: Reduce number/size of signals in DAQ lists or increase event period; confirm max DTO via `xcp-info`.
- A2L mismatch: Ensure the A2L matches the firmware build; symbolic access depends on correct addresses and conversions.

5.9 Next Steps

- Explore the examples directory for more advanced usage patterns
- Check the API documentation for detailed information about available functions
- Join the community to get help and contribute to the project

CHAPTER

SIX

XCP + A2L INTEGRATION GUIDE

6.1 Overview

This guide demonstrates how to use **ASAM MCD-2 MC (A2L)** files with **pyXCP** for symbolic access to ECU parameters. A2L files describe the memory layout, data types, and conversion formulas of ECU variables, enabling human-readable measurement and calibration workflows.

What you'll learn:

1. Load A2L files with `pya2ldb`
2. Read measurements by symbolic name
3. Write calibration parameters by name
4. Set up DAQ with A2L metadata
5. Convert raw values to engineering units
6. Export data with symbolic labels

For **production-grade workflows** with advanced features, see the asamint project.

6.2 Tools Overview

6.2.1 pyXCP Ecosystem

6.2.2 When to Use What?

Use pyxcp directly when:

- Building custom calibration tools
- Integrating XCP into test automation
- Low-level protocol debugging
- Learning XCP fundamentals
- Embedded in larger applications

Use asamint when:

- Need command-line MCS functionality
- Orchestrating multiple ASAM standards (A2L + MDF + XCP)
- Creating calibration data files (ASAM CDF)
- High-level batch operations

- Production measurement campaigns

Example decision tree:

```
Need XCP communication?
└── YES
    ├── Simple script / learning? → Use pyxcp examples
    ├── Custom tool / integration? → Use pyxcp API directly
    └── Production MCS? → Use asamint (built on pyxcp)
```

6.3 Prerequisites

Install required packages:

```
pip install pyxcp pya2ldb
```

Optional (for asamint):

```
git clone https://github.com/christoph2/asamint
cd asamint
python setup.py develop
```

6.4 Quick Start: Read Parameter by Name

Scenario: Read ECU software version string using A2L symbolic name.

```python
from pyxcp.cmdline import ArgumentParser
from pya2ldb import DB

# Load A2L file
db = DB()
db.import_a2l("my_ecu.a2l")

# Get measurement metadata
version_var = db.query_measurement("SwVersion")
address = version_var.address
datatype = version_var.datatype  # e.g., "UBYTE[16]"

# Connect to ECU
ap = ArgumentParser(description="Read SW version")
with ap.run() as xcp:
    xcp.connect()

    # Read by address (from A2L)
    raw_data = xcp.fetch(address, length=16)
    version = raw_data.decode('utf-8').rstrip('\x00')

    print(f"Software Version: {version}")

    xcp.disconnect()
```

6.5 Complete Workflow: Calibration with A2L

This example demonstrates a full calibration cycle:

1. Load A2L database
2. Query characteristics (calibration parameters)
3. Read current values
4. Modify parameters
5. Write back to ECU
6. Verify changes

```
#!/usr/bin/env python
"""Complete A2L-based calibration workflow."""

from pyxcp.cmdline import ArgumentParser
from pya2ldb import DB
import struct

# === Configuration ===
A2L_FILE = "my_ecu.a2l"
PARAM_NAME = "InjectionTiming"  # Characteristic to calibrate

# Load A2L
db = DB()
db.import_a2l(A2L_FILE)

# Get parameter metadata
param = db.query_characteristic(PARAM_NAME)
address = param.address
conversion = param.conversion  # e.g., RAT_FUNC with formula

# Connect to ECU
ap = ArgumentParser(description=f"Calibrate {PARAM_NAME}")
with ap.run() as xcp:
    xcp.connect()

    # 1. Read current value (raw)
    raw_bytes = xcp.fetch(address, length=param.size)
    raw_value = struct.unpack(param.format_string, raw_bytes)[0]

    # 2. Convert to physical value (engineering units)
    if conversion:
        physical_value = conversion.raw_to_phys(raw_value)
    else:
        physical_value = raw_value

    print(f"Current {PARAM_NAME}: {physical_value} {param.unit}")

    # 3. Modify parameter
    new_physical = physical_value * 1.05  # 5% increase
```

(continues on next page)

(continued from previous page)

```
    # 4. Convert back to raw value
    if conversion:
        new_raw = conversion.phys_to_raw(new_physical)
    else:
        new_raw = new_physical

    new_bytes = struct.pack(param.format_string, int(new_raw))

    # 5. Write to ECU
    xcp.download(address, new_bytes)
    print(f"Updated {PARAM_NAME}: {new_physical} {param.unit}")

    # 6. Verify
    verify_bytes = xcp.fetch(address, length=param.size)
    verify_raw = struct.unpack(param.format_string, verify_bytes)[0]
    verify_phys = conversion.raw_to_phys(verify_raw) if conversion else verify_raw

    assert abs(verify_phys - new_physical) < 0.01, "Verification failed!"
    print("✓ Verification passed")

    xcp.disconnect()
```

6.6 DAQ Setup with A2L Metadata

Use A2L file to automatically configure DAQ lists with symbolic names:

```
from pyxcp.cmdline import ArgumentParser
from pyxcp.daq_stim import DaqList, DaqToCsv
from pya2ldb import DB

# Load A2L
db = DB()
db.import_a2l("my_ecu.a2l")

# Define measurements to record
measurements = ["EngineSpeed", "VehicleSpeed", "Throttle", "CoolantTemp"]

# Build ODT entries from A2L
odt_entries = []
for name in measurements:
    meas = db.query_measurement(name)
    odt_entries.append({
        "address": meas.address,
        "size": meas.size,
        "name": name,
        "unit": meas.unit,
        "datatype": meas.datatype
    })

# Connect and setup DAQ
ap = ArgumentParser(description="DAQ from A2L")
```

(continues on next page)

(continued from previous page)

```
with ap.run() as xcp:
    xcp.connect()

    # Allocate DAQ list
    daq = DaqList(xcp, 0, event_channel=0)

    # Add ODTs from A2L metadata
    for entry in odt_entries:
        daq.add_odt_entry(
            address=entry["address"],
            size=entry["size"]
        )

    # Start recording
    csv_writer = DaqToCsv("recording.csv", header=[e["name"] for e in odt_entries])

    xcp.startDaq(daq.daq_list_number)

    # Collect 100 samples
    for _ in range(100):
        data = xcp.daqQueue.get(timeout=1.0)
        csv_writer.write_row(data)

    xcp.stopDaq()
    csv_writer.close()

    print("✓ Recording saved to recording.csv with symbolic names")

    xcp.disconnect()
```

See the complete example at: `pyxcp/examples/daq_from_a2l.py`

6.7 A2L File Structure

Understanding A2L structure helps troubleshoot issues:

```
/begin PROJECT MyProject
  /begin MODULE ECU_Controller

    /begin MEASUREMENT EngineSpeed  "Engine RPM"
      UWORD 0x4000  /* address */
      RAT_FUNC 1.0 0.0  /* factor, offset */
      0.0 8000.0  /* min, max */
      "rpm"  /* unit */
    /end MEASUREMENT

    /begin CHARACTERISTIC InjectionTiming  "Fuel injection timing"
      VALUE 0x5000  /* address */
      SWORD  /* datatype */
      RAT_FUNC 0.01 0.0  /* factor, offset */
      -50.0 50.0  /* min, max */
      "deg"  /* unit */
```

(continues on next page)

(continued from previous page)

```
    /end CHARACTERISTIC

  /end MODULE
/end PROJECT
```

Key sections:

- **MEASUREMENT**: Read-only variables (sensors, status)
- **CHARACTERISTIC**: Calibration parameters (maps, curves, scalars)
- **COMPU_METHOD**: Conversion formulas (RAT_FUNC, TAB_VERB, etc.)
- **IF_DATA XCP**: XCP-specific configuration (addresses, DAQ setup)

6.8 Data Type Conversion

A2L defines conversion methods for raw physical values:

6.8.1 RAT_FUNC (Rational Function)

Most common conversion:

```
physical = (raw_value * factor) + offset
```

Example:

```
# A2L: RAT_FUNC 0.1 -40.0
raw_value = 250
physical = (250 * 0.1) + (-40.0)  # = -15.0 °C
```

6.8.2 TAB_VERB (Table Interpolation)

For non-linear conversions:

```
/begin COMPU_METHOD LookupTable
  TAB_VERB "Lookup table"
  /begin COMPU_TAB_REF
    DEFAULT_VALUE "N/A"
    /begin VALUES
      0   0.0
      50  12.5
      100 25.0
      255 100.0
    /end VALUES
  /end COMPU_TAB_REF
/end COMPU_METHOD
```

6.9 Working Example

Complete working example is available at:

pyxcp/examples/a2l_integration.py

Features:

- Loading A2L files
- Querying measurements and characteristics
- Reading parameters by name
- Writing calibration values
- DAQ setup with symbolic names
- CSV export with headers

Run it:

```
python pyxcp/examples/a2l_integration.py --transport CAN --channel 0
```

6.10 Advanced: Production Workflows with asamint

For production-grade measurement and calibration, use **asamint**:

Key features:

1. **Command-line MCS**: No GUI needed
2. **Batch operations**: Automate calibration campaigns
3. **MDF export**: Industry-standard format (ASAM MCD-3 MC)
4. **CDF creation**: Generate calibration data files
5. **Multiple projects**: Orchestrate pyxcp, pya2ldb, asammdf, objutils

Example: Create CDF from XCP slave

```
# asamint example (command-line MCS)
asamint create-cdf my_ecu.a2l --output calibration.cdf
```

Example: Setup dynamic DAQ with MDF output

```
# Using asamint API
from asamint import Session

session = Session("my_ecu.a2l")
session.connect("CAN", channel=0)

# High-level API
session.setup_daq(["Speed", "Torque", "Temp"])
session.record_mdf("recording.mdf", duration=60)

session.disconnect()
```

Learn more:

- Repository: https://github.com/christoph2/asamint
- Examples: `asamint/examples/` directory
- Documentation: `asamint/docs/`

6.11 Troubleshooting

6.11.1 Common A2L Issues

"Measurement not found"

- Check symbolic name spelling (case-sensitive!)
- Verify A2L file version matches ECU firmware
- Use `db.list_measurements()` to list all available

"Address access error"

- A2L address might be incorrect (wrong firmware version)
- ECU might require SEED/KEY unlock first
- Memory protection: use `xcp.setCalPage()` if needed

"Conversion failed"

- A2L might have invalid COMPU_METHOD
- Raw value out of bounds (check min/max)
- Use `meas.conversion` to inspect formula

"DAQ configuration error"

- ODT size exceeds max_dto (check A2L IF_DATA)
- Event channel not supported (query with `xcp.getDaqEventInfo()`)
- Too many ODT entries (check DAQ limits)

6.11.2 Performance Tips

1. **Batch reads**: Use `xcp.upload(address, size)` for multiple variables
2. **DAQ for monitoring**: Prefer DAQ over polling for high-rate signals
3. **Cache A2L database**: Don't reload A2L on every operation
4. **XCP master lock**: Use locking for multi-threaded calibration

6.12 FAQ

Q: Can I use pyxcp without A2L files?

Yes! pyxcp works with raw addresses. A2L provides symbolic access convenience.

Q: What's the difference between pya2ldb and pya2l?

- `pya2l`: Legacy parser (deprecated)
- `pya2ldb`: Modern database-backed parser (recommended)

Q: Does pyxcp generate A2L files?

No. A2L files are generated by ECU development tools (INCA, CANape, etc.).

Q: When should I use asamint instead of pyxcp?

Use asamint for:

- Command-line batch operations

- MDF output (industry standard)
- Orchestrating multiple ASAM tools
- Production measurement campaigns

Use pyxcp for:

- Custom Python applications
- Test automation
- Embedded in larger systems
- Learning XCP protocol

Q: Can I edit A2L files?

Technically yes (they're text files), but **not recommended**. A2L files are generated from ECU source code and should stay in sync with firmware.

Q: What if my ECU doesn't have an A2L file?

You'll need to:

1. Get A2L from ECU supplier
2. Reverse-engineer memory layout (advanced!)
3. Use XCP with raw addresses only

6.13 Related Examples

See `pyxcp/examples/` directory:

- `a2l_integration.py`: Complete A2L workflow
- `daq_from_a2l.py`: DAQ setup from A2L
- `calibration_workflow.py`: Raw address calibration
- `daq_recording.py`: DAQ recording basics

6.14 References

- **ASAM MCD-2 MC (A2L) Standard**: https://www.asam.net/standards/detail/mcd-2-mc/
- **pya2ldb Repository**: https://github.com/christoph2/pya2l
- **asamint Repository**: https://github.com/christoph2/asamint
- **pyxcp Repository**: https://github.com/christoph2/pyxcp
- **ASAM Standards**: https://www.asam.net/standards/

Next steps:

- Read *pyXCP Tutorial* for pyxcp basics
- Check *Configuration* for advanced XCP setup
- Try ../examples/a2l_integration for hands-on practice

CHAPTER

SEVEN

CONFIGURATION

pyXCP supports various configuration options for different transport layers and use cases. This guide explains how to configure pyXCP for your specific needs.

7.1 Configuration Systems

pyXCP supports two configuration systems:

1. **Traitlets-based Configuration (Recommended)**: Python-based configuration using the traitlets library
2. **Legacy Configuration**: TOML or JSON based configuration (deprecated, don't use in new project)

7.2 Traitlets-based Configuration

The recommended way to configure pyXCP is using Python configuration files with the traitlets system:

```
# Configuration file for pyXCP
c = get_config()  # noqa

# Transport configuration
c.Transport.layer = "ETH"  # Transport layer: ETH, CAN, USB, SXI

# Ethernet configuration
c.Transport.Eth.host = "localhost"
c.Transport.Eth.port = 5555
c.Transport.Eth.protocol = "TCP"
```

You can generate a template configuration file with all available options:

```
xcp-profile create -o my_config.py
```

7.3 Legacy Configuration (Deprecated)

The older TOML and JSON based configuration is still supported but is now considered legacy:

```
[XCP]
# General XCP settings
TRANSPORT = "ETH"  # Transport layer: ETH, CAN, USB, SXI

[ETH]
```

(continues on next page)

(continued from previous page)

```
# Transport-specific settings
HOST = "localhost"
PORT = 5555
```

You can convert a legacy configuration file to the new format:

```
xcp-profile convert -c old_config.toml -o new_config.py
```

7.4 Transport Layer Configuration

7.4.1 Ethernet (TCP/IP)

Recommended Python configuration:

```
# Transport configuration
c.Transport.layer = "ETH"

# Ethernet configuration
c.Transport.Eth.host = "localhost"  # IP address or hostname
c.Transport.Eth.port = 5555         # Port number
c.Transport.Eth.protocol = "TCP"    # TCP or UDP
```

Legacy TOML configuration:

```
[XCP]
TRANSPORT = "ETH"

[ETH]
HOST = "localhost"  # IP address or hostname
PORT = 5555         # Port number
PROTOCOL = "TCP"    # TCP or UDP
LOGLEVEL = "INFO"   # Optional: DEBUG, INFO, WARNING, ERROR, CRITICAL
```

7.4.2 CAN

Recommended Python configuration:

```
# Transport configuration
c.Transport.layer = "CAN"

# CAN configuration
c.Transport.Can.interface = "vector"  # CAN interface supported by python-can
c.Transport.Can.channel = 0           # Channel identification
c.Transport.Can.bitrate = 500000      # CAN bitrate in bits/s
c.Transport.Can.can_id_master = 0x7E0 # CAN-ID master -> slave
c.Transport.Can.can_id_slave = 0x7E1  # CAN-ID slave -> master
c.Transport.Can.max_dlc_required = False # Master to slave frames always to have DLC =␣
↪MAX_DLC = 8
```

Legacy TOML configuration:

```
[XCP]
TRANSPORT = "CAN"

[CAN]
INTERFACE = "vector"
CHANNEL = 0
BITRATE = 500000
CAN_ID_MASTER = 2016
CAN_ID_SLAVE = 2017
```

7.4.3 USB

```
c.Transport.layer = "USB"
# Configure USB specifics as supported by your environment
```

7.4.4 SXI

```
c.Transport.layer = "SXI"
# Configure SXI specifics as needed
```

7.5 Additional Notes

- Prefer Python/traitlets configuration for new projects.
- Use `xcp-profile create` to bootstrap a config and `xcp-profile convert` to migrate legacy TOML/JSON.
- Ensure the chosen transport layer matches any externally provided interface (e.g., if passing a pre-created CAN interface, set `c.Transport.layer = 'CAN'`).

CHAPTER

EIGHT

LOGGING CONFIGURATION

8.1 Overview

pyxcp uses Python's standard `logging` module with a **NullHandler by default** to avoid interfering with your application's logging configuration.

This follows Python's logging best practices for libraries: **libraries should never configure logging**, only applications should.

8.2 Default Behavior

By default, pyxcp produces **no log output**:

```
from pyxcp.cmdline import ArgumentParser

ap = ArgumentParser(description="Silent by default")
with ap.run() as xcp:
    xcp.connect()
    # No logging output unless you configure it
    xcp.disconnect()
```

This ensures pyxcp doesn't interfere with your own logging setup.

8.3 Logger Hierarchy

pyxcp uses a hierarchical logger structure:

```
pyxcp                          # Root logger (NullHandler)
├── pyxcp.master               # Master class logs
│   └── pyxcp.master.errorhandler  # Error handling logs
├── pyxcp.transport            # Transport layer logs
├── pyxcp.daq_stim             # DAQ/STIM logs
└── pyxcp.recorder.converter   # Recorder logs
```

All loggers inherit from the pyxcp root logger, so configuring the root affects all child loggers.

8.4 Enabling Logging

8.4.1 Method 1: Quick Setup (Recommended)

Use the built-in `setup_logging()` helper:

```
from pyxcp.logger import setup_logging
import logging

# Enable INFO logging to console
setup_logging(level=logging.INFO)

# Now pyxcp logs will appear
from pyxcp.cmdline import ArgumentParser
ap = ArgumentParser(description="With logging")
with ap.run() as xcp:
    xcp.connect()  # Will log connection details
    xcp.disconnect()
```

8.4.2 Method 2: Standard Python Logging

Configure using Python's standard logging:

```
import logging

# Configure root logger for your application
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(name)s] %(levelname)s: %(message)s"
)

# pyxcp logs will now appear with 'pyxcp.*' names
from pyxcp.cmdline import ArgumentParser
ap = ArgumentParser(description="Standard logging")
with ap.run() as xcp:
    xcp.connect()
    xcp.disconnect()
```

8.4.3 Method 3: File Logging

Log to file instead of console:

```
import logging

# Create file handler
handler = logging.FileHandler("pyxcp_session.log", mode="w")
formatter = logging.Formatter(
    "[%(asctime)s] [%(name)s] %(levelname)s: %(message)s"
)
handler.setFormatter(formatter)

# Configure pyxcp logger
pyxcp_logger = logging.getLogger("pyxcp")
```

(continues on next page)

(continued from previous page)

```
pyxcp_logger.addHandler(handler)
pyxcp_logger.setLevel(logging.DEBUG)

# All pyxcp activity logged to file
from pyxcp.cmdline import ArgumentParser
ap = ArgumentParser(description="File logging")
with ap.run() as xcp:
    xcp.connect()
    xcp.disconnect()
```

8.4.4 Method 4: Selective Module Logging

Enable logging for specific pyxcp modules only:

```
import logging

logging.basicConfig(level=logging.WARNING)  # App default: warnings only

# But enable DEBUG for transport layer only
transport_logger = logging.getLogger("pyxcp.transport")
transport_logger.setLevel(logging.DEBUG)

# Now only transport logs at DEBUG, rest at WARNING
from pyxcp.cmdline import ArgumentParser
ap = ArgumentParser(description="Selective logging")
with ap.run() as xcp:
    xcp.connect()  # Transport DEBUG logs appear
    xcp.disconnect()
```

8.5 Advanced Configuration

8.5.1 Multiple Handlers

Send logs to multiple destinations:

```
import logging

pyxcp_logger = logging.getLogger("pyxcp")
pyxcp_logger.setLevel(logging.DEBUG)

# Console handler (INFO and above)
console = logging.StreamHandler()
console.setLevel(logging.INFO)
console.setFormatter(logging.Formatter("%(levelname)s: %(message)s"))
pyxcp_logger.addHandler(console)

# File handler (DEBUG and above)
file_handler = logging.FileHandler("pyxcp_debug.log")
file_handler.setLevel(logging.DEBUG)
file_handler.setFormatter(logging.Formatter(
    "%(asctime)s [%(name)s] %(levelname)s: %(message)s"
```

(continues on next page)

(continued from previous page)

```
))
pyxcp_logger.addHandler(file_handler)

# INFO+ to console, DEBUG+ to file
```

8.5.2 Custom Formatting

Use custom log format for pyxcp:

```
import logging

# Custom format with milliseconds and function name
formatter = logging.Formatter(
    "%(asctime)s.%(msecs)03d [%(name)s:%(funcName)s] %(levelname)s: %(message)s",
    datefmt="%H:%M:%S"
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)

pyxcp_logger = logging.getLogger("pyxcp")
pyxcp_logger.addHandler(handler)
pyxcp_logger.setLevel(logging.DEBUG)
```

8.5.3 Integration with Existing Loggers

If your application already has logging configured, pyxcp will inherit it:

```
import logging

# Your application's logging setup
logging.basicConfig(
    level=logging.INFO,
    format="[%(levelname)s] %(name)s: %(message)s",
    handlers=[
        logging.FileHandler("app.log"),
        logging.StreamHandler()
    ]
)

# pyxcp logs will automatically use your configuration
from pyxcp.cmdline import ArgumentParser
ap = ArgumentParser(description="Integrated logging")
with ap.run() as xcp:
    xcp.connect()  # Logs to app.log and console
    xcp.disconnect()
```

8.6 Logging Levels

pyxcp uses standard Python logging levels:

Level	Usage
DEBUG	Detailed diagnostic information (frame contents, state transitions)
INFO	General informational messages (connection established, DAQ started)
WARNING	Something unexpected but not critical (timeout, retry)
ERROR	Error occurred but operation continues (command failed)
CRITICAL	Serious error, operation cannot continue

Recommended levels:

- **Production**: `WARNING` or `ERROR` only
- **Development**: `INFO` for general visibility
- **Debugging**: `DEBUG` for detailed protocol traces

8.7 Troubleshooting

8.7.1 "No logs appearing"

1. **Check handler is configured**:

```
import logging
pyxcp_logger = logging.getLogger("pyxcp")
print(f"Handlers: {pyxcp_logger.handlers}")  # Should not be empty
```

2. **Check log level**:

```
import logging
pyxcp_logger = logging.getLogger("pyxcp")
print(f"Level: {logging.getLevelName(pyxcp_logger.level)}")
```

3. **Check propagation**:

```
import logging
pyxcp_logger = logging.getLogger("pyxcp")
print(f"Propagate: {pyxcp_logger.propagate}")  # Should be True
```

8.7.2 "ValueError: I/O operation on closed file" (Issue #176)

Cause: Your application's file handler was closed while pyxcp tried to log.

Solution: Ensure proper shutdown order:

```
import logging

# Your logging setup
file_handler = logging.FileHandler("myapp.log")
logger = logging.getLogger("myapp")
logger.addHandler(file_handler)
```

(continues on next page)

(continued from previous page)

```
# Use pyxcp
from pyxcp.cmdline import ArgumentParser
with ap.run() as xcp:
    xcp.connect()
    xcp.disconnect()

# Close file handler AFTER pyxcp is done
file_handler.close()
logger.removeHandler(file_handler)
```

8.7.3 "Logs duplicated"

Cause: Multiple handlers registered or propagation issues.

Solution: Clear existing handlers before reconfiguring:

```
import logging

pyxcp_logger = logging.getLogger("pyxcp")

# Clear existing handlers
for handler in pyxcp_logger.handlers[:]:
    pyxcp_logger.removeHandler(handler)

# Add your handler
new_handler = logging.StreamHandler()
pyxcp_logger.addHandler(new_handler)
```

8.8 Examples

8.8.1 Minimal Debug Session

```
from pyxcp.logger import setup_logging
from pyxcp.cmdline import ArgumentParser
import logging

# Quick debug setup
setup_logging(level=logging.DEBUG)

ap = ArgumentParser(description="Debug session")
with ap.run() as xcp:
    xcp.connect()
    result = xcp.fetch(0x1000, 4)
    print(f"Data: {result.hex()}")
    xcp.disconnect()
```

8.8.2 Production with File Rotation

```
import logging
from logging.handlers import RotatingFileHandler
```

(continues on next page)

(continued from previous page)

```
# Rotating file handler (10 MB max, 5 backups)
handler = RotatingFileHandler(
    "pyxcp.log",
    maxBytes=10*1024*1024,
    backupCount=5
)
formatter = logging.Formatter(
    "%(asctime)s [%(name)s] %(levelname)s: %(message)s"
)
handler.setFormatter(formatter)

pyxcp_logger = logging.getLogger("pyxcp")
pyxcp_logger.addHandler(handler)
pyxcp_logger.setLevel(logging.WARNING)  # Production: warnings only
```

8.8.3 Multi-Application Integration

```
import logging

# Application logger
app_logger = logging.getLogger("myapp")
app_logger.setLevel(logging.INFO)

# pyxcp logger (child of root)
pyxcp_logger = logging.getLogger("pyxcp")
pyxcp_logger.setLevel(logging.WARNING)  # Less verbose than app

# Shared handler
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter(
    "%(asctime)s [%(name)s] %(levelname)s: %(message)s"
))

# Add to both
app_logger.addHandler(handler)
# pyxcp inherits from root, so configure root too
logging.root.addHandler(handler)
logging.root.setLevel(logging.INFO)
```

8.9 FAQ

Q: Why doesn't pyxcp log anything by default?

A: Following Python best practices, libraries should not configure logging. Only applications should. This prevents pyxcp from interfering with your logging setup.

Q: How do I enable verbose logging for debugging?

A: Use `setup_logging(level=logging.DEBUG)` from `pyxcp.logger`.

Q: Can I use different log levels for different pyxcp modules?

A: Yes! Configure loggers like `logging.getLogger("pyxcp.transport").setLevel(logging.DEBUG)`.

Q: Does pyxcp use print() statements?

A: No. All output uses proper logging (except for CLI tools which use `rich.console`).

Q: Can I disable logging completely?

A: It's already disabled by default (NullHandler). If you see logs, it's because you or another library configured logging.

Q: How do I log to syslog/network/database?

A: Use Python's standard logging handlers (`SysLogHandler`, `SocketHandler`, etc.) with the `pyxcp` logger.

8.10 Related Documentation

- troubleshooting - Common errors and solutions
- *Configuration* - Configuration system
- Python logging documentation: https://docs.python.org/3/library/logging.html
- Logging HOWTO: https://docs.python.org/3/howto/logging.html

8.11 References

Issue #176: User logging configuration conflict

Python logging best practices for libraries:

- Use NullHandler by default
- Never call `logging.basicConfig()` in library code
- Use hierarchical logger names
- Let applications configure logging

CHAPTER

NINE

FRAME ACQUISITION POLICY MIGRATION GUIDE

9.1 Overview

pyXCP uses **Frame Acquisition Policies** to handle incoming XCP frames. As of version 0.27.0, the default policy has changed from `LegacyFrameAcquisitionPolicy` to `NoOpPolicy` to prevent unbounded memory growth.

9.2 Why the Change?

`LegacyFrameAcquisitionPolicy` suffers from unbounded memory leaks in long-running DAQ sessions:

- **8 unbounded C++ queues** (CMD, RES, EV, SERV, DAQ, META, ERR, STIM)
- **Memory growth**: ~23 MB/hour at 100 Hz DAQ rate
- **24-hour leak**: Nearly 2 GB of accumulated frames
- **Root cause**: Event queue (`evQueue`) grows indefinitely

The policy is explicitly marked as **deprecated** in the C++ code:

```
/*
    Dequeue based frame acquisition policy.

    Deprecated: Use only for compatibility reasons.
*/
```

9.3 Available Policies

9.3.1 NoOpPolicy (Default)

Discards all frames immediately - No memory footprint, ideal for most use cases where frames are processed in real-time through callbacks.

```
from pyxcp.transport.base import NoOpPolicy

policy = NoOpPolicy(filtered_out=None)
with ap.run(policy=policy) as x:
    x.connect()
    # Your code here
```

Memory: O(1) - No accumulation

Use case: Standard XCP command/response, DAQ with real-time processing

9.3.2 FrameRecorderPolicy (Recommended for Recording)

Streams frames directly to disk in `.xmraw` format - Constant memory usage, high performance.

```
from pyxcp.transport.base import FrameRecorderPolicy

policy = FrameRecorderPolicy("session.xmraw", filtered_out=None)
with ap.run(policy=policy) as x:
    x.connect()
    # DAQ recording session
    x.disconnect()
```

Memory: O(1) - Frames written to disk immediately

Use case: Long-running DAQ recording, post-processing with `xmraw-converter`

9.3.3 StdoutPolicy (Debug/Development)

Prints frames to console - Useful for debugging transport-layer issues.

```
from pyxcp.transport.base import StdoutPolicy

policy = StdoutPolicy(filtered_out=None)
with ap.run(policy=policy) as x:
    x.connect()
    # Frames printed to stdout
```

Memory: O(1) - Frames printed immediately

Use case: Debugging, development, protocol analysis

9.3.4 PyFrameAcquisitionPolicy (Custom)

Python callback-based - Maximum flexibility for custom frame processing.

```
from pyxcp.transport.transport_ext import FrameAcquisitionPolicy, FrameCategory

class MyPolicy(FrameAcquisitionPolicy):
    def __init__(self):
        super().__init__(filtered_out=None)
        self.daq_count = 0

    def feed(self, frame_category, counter, timestamp, payload):
        if frame_category == FrameCategory.DAQ:
            self.daq_count += 1
            # Process DAQ frame

    def finalize(self):
        print(f"Processed {self.daq_count} DAQ frames")

policy = MyPolicy()
with ap.run(policy=policy) as x:
    x.connect()
    # ...
```

Memory: Depends on implementation

Use case: Real-time analytics, custom filtering, live dashboards

9.3.5 LegacyFrameAcquisitionPolicy (Deprecated)

DO NOT USE for new code. Retained only for backward compatibility.

```
#  DEPRECATED - Causes memory leaks!
from pyxcp.transport.base import LegacyFrameAcquisitionPolicy

policy = LegacyFrameAcquisitionPolicy(filtered_out=None)
# DeprecationWarning will be emitted
```

Memory: O(n) - **Unbounded growth** over time

Known issues: - Event queue grows without limit - 24-hour DAQ session can leak ~2 GB - Queues never automatically consumed

9.4 Migration Checklist

If you have existing code using `LegacyFrameAcquisitionPolicy`:

1. **Identify your use case**:
   - Recording DAQ data → `FrameRecorderPolicy`
   - Real-time processing → `FrameAcquisitionPolicy`
   - Standard operation → `NoOpPolicy` (default)
   - Debugging → `StdoutPolicy`
2. **Update policy instantiation**:

   ```
   # Before (deprecated)
   from pyxcp.transport.base import LegacyFrameAcquisitionPolicy
   policy = LegacyFrameAcquisitionPolicy(filtered_out=None)

   # After (recommended)
   from pyxcp.transport.base import FrameRecorderPolicy
   policy = FrameRecorderPolicy("recording.xmraw", filtered_out=None)
   ```

3. **Remove queue access**:

   Legacy policy exposed queues (`resQueue`, `daqQueue`, etc.). Modern policies use:
   - **Callbacks**: `FrameAcquisitionPolicy.feed()`
   - **Files**: `FrameRecorderPolicy` → read with `XcpLogFileReader`
   - **Real-time**: Process in DAQ callback, not via queues
4. **Test memory usage**:

   ```
   # Run your benchmark
   python pyxcp/benchmarks/daq_memory_test.py

   # Should show stable memory (~0 MB/s growth)
   ```

9.5 Performance Comparison

Validation Tests (5-minute DAQ simulation @ 100 Hz, 30,000 frames):

Key findings:

- **NoOpPolicy**: 290x reduction in memory leak vs. Legacy (6.75 MB vs 1,958 MB in 24h)
- **FrameRecorderPolicy**: 25x reduction + data persisted to disk
- **Legacy Policy**: Confirmed unbounded growth, ~2 GB leak in 24h @ 100 Hz DAQ

Validation methodology:

- Test duration: 5 minutes (300 seconds)
- DAQ rate: 100 Hz (simulated)
- Frames processed: ~27,500-28,000
- Memory sampled every 60 seconds
- Python 3.13, psutil memory tracking
- Validation scripts: `pyxcp/benchmarks/validation_*.py`

Backward Compatibility

Default behavior change:

- **Before v0.27.0**: `LegacyFrameAcquisitionPolicy()` (implicit)
- **After v0.27.0**: `NoOpPolicy(filtered_out=None)` (implicit)

Breaking change: Code that relied on accessing `.daqQueue` or `.evQueue` from the policy will break. Use explicit policy:

```
# Temporary compatibility (not recommended)
from pyxcp.transport.base import LegacyFrameAcquisitionPolicy
import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)
policy = LegacyFrameAcquisitionPolicy(filtered_out=None)
```

Long-term fix: Migrate to modern callback-based approach.

9.6 Examples

See `pyxcp/examples/xcp_policy.py` for working examples of all policies.

For recording workflows, see `pyxcp/examples/daq_recording.py`.

9.7 Related Issues

- #171 - Memory leak in DAQ mode
- #218 - Performance optimization opportunities

9.8 References

- C++ implementation: `pyxcp/transport/transport_ext.hpp`
- Python bindings: `pyxcp/transport/transport_wrapper.cpp`
- Recorder format: `pyxcp/recorder/` (xmraw specification)

CHAPTER

TEN

CUSTOM CAN BUS OBJECTS

By default, pyXCP automatically creates and manages CAN bus objects for you. In most situations this means you don't need to worry about setting up the bus manually.

There are three levels of CAN interface integration, ordered from simplest to most flexible:

1. **python-can plugin interface** — use any driver that registers itself via python-can's entry-point plugin system (no wrapper code).
2. **Existing bus object** — pass a pre-created `can.Bus` instance directly (e.g. shared with another application such as `UDSonCAN`).
3. **Fully custom interface** — implement `CanInterfaceBase` for hardware that is not supported by python-can at all.

10.1 python-can plugin interface (recommended for third-party drivers)

python-can ships with a plugin interface that lets external packages register additional CAN drivers via Python entry points. Once such a package is installed, its driver is available to *both* `can.Bus()` and — directly to pyxcp — without any wrapper code.

Simply set `c.Transport.Can.interface` to the plugin's registered name, exactly as you would with `can.Bus(interface="...")`.

Example: python-can-remote

python-can-remote provides a CAN-over-network bridge and registers under the name `"remote"`. Install it with:

```
pip install python-can-remote
```

Then configure pyxcp:

```
# pyxcp_conf.py
c.Transport.layer = "CAN"
c.Transport.Can.interface     = "remote"           # python-can plugin name
c.Transport.Can.channel       = "ws://myhost:54701/"
c.Transport.Can.bitrate       = 500_000
c.Transport.Can.can_id_master = 0x01
c.Transport.Can.can_id_slave  = 0x02
```

pyxcp forwards the common parameters (`channel`, `bitrate`, `fd`, `data_bitrate`, `receive_own_messages`, `poll_interval`) to python-can, which resolves the actual driver via its plugin registry.

10.1.1 Passing driver-specific parameters (`extra_params`)

If the driver requires keyword arguments beyond the common base set, use `c.Transport.Can.CanCustom.extra_params`. Every entry is merged verbatim into the `can.Bus()` call *after* the standard parameters (and therefore takes precedence on key overlap):

```
# pyxcp_conf.py
c.Transport.layer = "CAN"
c.Transport.Can.interface = "remote"
c.Transport.Can.channel   = "ws://myhost:54701/"
c.Transport.Can.bitrate   = 500_000

# Driver-specific extras forwarded verbatim to can.Bus():
c.Transport.Can.CanCustom.extra_params = {"rx_queue_size": 128, "proprietary_option":␣
↪4711}
```

10.1.2 Known python-can plugin packages

The following packages are maintained by third parties. Issues should be reported in their respective repositories.

Package / interface name	Description
`python-can-remote` / `"remote"`	CAN over network bridge
`python-can-canine` / `"canine"`	CAN Driver for the CANine CAN interface
`python-can-cvector` / `"cvector"`	Cython-based version of the Vector bus
`python-can-sontheim` / `"sontheim"`	CAN Driver for Sontheim interfaces (e.g. CANfox)
`zlgcan` / `"zlgcan"`	Python wrapper for zlgcan-driver-rs
`python-can-cando` / `"cando"`	Python wrapper for Netronics' CANdo and CANdoISO
`python-can-candle` / `"candle"`	Full-featured driver for candleLight

Note

Any python-can plugin that correctly registers a `can.interface` entry point will work — the list above is not exhaustive. See the python-can plugin documentation for details on writing your own plugin.

10.2 Using an existing CAN bus object

If you already hold a `can.Bus` instance (e.g. shared with `UDSonCAN` or another tool running in the same process), pass it directly via the `transport_layer_interface` parameter:

```
import can
from pyxcp.cmdline import ArgumentParser

can_if = can.Bus(interface="kvaser", channel="0", fd=False, bitrate=500000)

ap = ArgumentParser(description="external interface test")
with ap.run(transport_layer_interface=can_if) as x:
    x.connect()
    x.disconnect()
```

> **Note**
>
> - It is the user's responsibility to properly initialize and shut down the CAN bus interface.
> - pyxcp **merges** its filter configuration with any existing one, so make sure no unwanted traffic is forwarded to other applications. The original filter configuration is restored when pyxCP exits.

10.3 Fully custom interface (hardware not supported by python-can)

If your hardware is not supported by python-can at all, implement the thin `CanInterfaceBase` ABC (a subset of python-can's BusABC):

```
from pyxcp.transport.can import CanInterfaceBase
```

Import the base class and implement the required abstract methods:

```
#!/usr/bin/env python
import can
from pyxcp.cmdline import ArgumentParser
from pyxcp.transport.can import CanInterfaceBase


class WrappedKvaserInterface(CanInterfaceBase):

    def __init__(self):
        self.canif = can.Bus(interface="kvaser", channel="0", bitrate=500000)

    def set_filters(self, filters):
        self.canif.set_filters(filters)

    def recv(self, timeout=None):
        return self.canif.recv(timeout)

    def send(self, msg):
        self.canif.send(msg)

    @property
    def filters(self):
        return self.canif.filters

    @property
    def state(self):
        return self.canif.state

    def close(self):
        self.canif.shutdown()


custom_interface = WrappedKvaserInterface()

ap = ArgumentParser(description="Wrapped Kvaser CAN driver.")
with ap.run(transport_layer_interface=custom_interface) as x:
```

(continues on next page)

(continued from previous page)

```
    x.connect()
    x.disconnect()

custom_interface.close()
```

A skeleton example is also available at pyxcp/examples/xcp_user_supplied_driver.py.

CHAPTER

ELEVEN

HOW-TOS

This page collects short guides and tips for common tasks.

- Using pyXCP Command-Line Tools
- How to build your own CAN drivers

CHAPTER

TWELVE

HOW-TO BUILD YOUR OWN CAN DRIVERS

By default, pyXCP automatically creates and manages CAN bus objects for you. In most situations, this means you don't need to worry about setting up the bus manually.

There are, however, two special cases where you might want to provide your own CAN bus object instead:

- You are already working with a CAN bus object in another application, such as UDSonCAN.
- You need to use an interface that is not supported by python-can.

In these situations, pyXCP allows you to integrate your existing setup rather than creating a new one.

12.1 Using an existing CAN bus object

```
import can
from pyxcp.cmdline import ArgumentParser

# Create your CAN bus object
can_if = can.Bus(interface="kvaser", channel="0", fd=False, bitrate=500000)

# Pass it to pyXCP
ap = ArgumentParser(description="external interface test")
with ap.run(transport_layer_interface=can_if) as x:
    x.connect()
    x.disconnect()
```

12.2 Using an unsupported interface

If your CAN interface is not supported by python-can, you can still integrate it by writing minimal wrapper code that provides the methods pyXCP requires (e.g. send() and recv()).

This is done by implementing the interface `CanInterfaceBase` (basically a subset of python-can's BusABC).

```
from pyxcp.transport.can import CanInterfaceBase
import can

class WrappedKvaserInterface(CanInterfaceBase):

    def __init__(self):
        self.canif = can.Bus(interface="kvaser", channel="0", bitrate=500000)

    def set_filters(self, filters):
```

(continues on next page)

(continued from previous page)

```
        self.canif.set_filters(filters)

    def recv(self, timeout: float = None):
        return self.canif.recv(timeout)

    def send(self, msg: can.message.Message):
        self.canif.send(msg)

    @property
    def filters(self):
        return self.canif.filters

    @property
    def state(self):
        return self.canif.state

    def close(self):
        self.canif.shutdown()


# Create your custom interface
custom_interface = WrappedKvaserInterface()

# Pass it to pyXCP
ap = ArgumentParser(description="Wrapped Kvaser CAN driver.")
with ap.run(transport_layer_interface=custom_interface) as x:
    x.connect()
    x.disconnect()

# Don't forget to close your interface
custom_interface.close()
```

12.3 Important Notes

- It is the user's responsibility to properly initialize and shut down the CAN bus interface.
- pyXCP merges its filter configuration with the existing one, so users must ensure that no unwanted traffic is passed to the external application. The original filter configuration is restored when pyXCP exits.
- Choose "custom" as your CAN interface in your configuration:

```
c.Transport.Can.interface = "custom"
```

CHAPTER

THIRTEEN

USING PYXCP COMMAND-LINE TOOLS

Note

New in v0.26.5+: For comprehensive CLI tools documentation, see *Command-Line Tools Reference*.

This page provides a quick overview. The new guide includes:

- Detailed usage for all 7 CLI tools
- Complete troubleshooting matrix
- Transport-specific tips
- Common workflows
- Environment variables

pyXCP provides several command-line tools to help you work with XCP devices. These tools are installed automatically when you install the pyXCP package.

13.1 Quick Reference

See *Command-Line Tools Reference* for detailed documentation of all tools.

13.2 Available Command-Line Tools

pyXCP includes the following command-line tools:

1. **pyxcp-probe-can-drivers**: Probes and lists available CAN drivers on your system.
2. **xcp-id-scanner**: Scans for XCP slaves on a CAN bus.
3. **xcp-fetch-a2l**: Fetches A2L file from an XCP slave.
4. **xcp-info**: Displays information about an XCP slave. It supports skipping certain categories of information (DAQ, PAG, PGM, IDs) to speed up the process or avoid issues with specific slaves.
5. **xcp-profile**: Creates new configuration files and converts legacy configuration files.
6. **xcp-examples**: Shows available examples and how to run them.
7. **xmraw-converter**: Converts XMRAW measurement files to other formats.

13.3 Basic Usage

All command-line tools follow a similar pattern for specifying the transport layer and connection parameters:

```
<tool-name> -t <transport> --config <config-file>
```

Where: - `<tool-name>` is one of the tools listed above - `<transport>` is the transport layer (eth, can, usb, sxi) - `<config-file>` is the path to a configuration file

13.4 Examples

Probe available CAN drivers:

```
pyxcp-probe-can-drivers
```

Display information about an XCP slave using Ethernet with Python configuration (recommended):

```
xcp-info -t eth --config conf_eth.py
```

You can also skip certain parts of the information gathering:

```
# Skip DAQ and PAG information
xcp-info -t eth --host 127.0.0.1 --port 5555 --no-daq --no-pag
```

Display information about an XCP slave using Ethernet with legacy TOML configuration:

```
xcp-info -t eth --config conf_eth.toml
```

Scan for XCP slaves on a CAN bus with Python configuration (recommended):

```
xcp-id-scanner -t can --config conf_can.py
```

Scan for XCP slaves on a CAN bus with legacy TOML configuration:

```
xcp-id-scanner -t can --config conf_can.toml
```

Convert an XMRAW file to CSV:

```
xmraw-converter measurement.xmraw -o csv
```

13.5 Using the xcp-profile Tool

The `xcp-profile` tool helps you manage configuration files for pyXCP. It supports two main use cases:

1. **create**: Generate a new Python-based configuration file with all available options
2. **convert**: Convert a legacy .json/.toml configuration file to the new Python-based format

13.5.1 Create a New Configuration

To create a new configuration file with all available options:

```
# Output to a file
xcp-profile create -o my_config.py

# Preview in terminal
xcp-profile create | less
```

CHAPTER

FOURTEEN

COMMAND-LINE TOOLS REFERENCE

pyXCP provides command-line tools for XCP device interaction, configuration management, and data conversion. All tools support the Python-based configuration system (v0.26.4+).

14.1 Quick Reference

- `xcp-info` – Inspect ECU capabilities
- `xcp-id-scanner` – Scan CAN bus for ECUs
- `xcp-fetch-a2l` – Download A2L from ECU
- `xcp-profile` – Create/convert configs
- `xcp-examples` – Copy example scripts
- `xcp-discovery` – Discover XCP-on-Ethernet slaves (multicast)
- `xmraw-converter` – Convert measurement data
- `pyxcp-probe-can-drivers` – List CAN drivers
- (extra) `xcp-daq-recorder` – Automated DAQ recording from JSON config

14.2 Configuration Methods

Method 1: Python config file (recommended):

```
xcp-info -t eth --config my_config.py
xcp-profile create -o my_config.py
```

Example transport factory:

```
# my_config.py
def create_transport(parent):
    from pyxcp.transport.eth import Eth
    return Eth(parent, host="192.168.1.100", port=5555, protocol="TCP")
```

Method 2: Command-line arguments:

```
xcp-info -t eth --host 192.168.1.100 --port 5555
xcp-info -t can --can-interface socketcan --can-channel can0
```

Method 3: Legacy TOML (deprecated, supported):

```
xcp-info -t eth --config legacy_config.toml
xcp-profile convert -o new_config.py old_config.toml
```

14.3 Tool Details

14.3.1 xcp-info

Inspect XCP slave capabilities (DAQ/PAG/PGM, IDs, protection).

Usage:

```
xcp-info [OPTIONS]
```

Common options: - `--no-daq` – skip DAQ queries - `--no-pag` – skip paging - `--no-pgm` – skip programming - `--no-ids` – skip ID scanning

Troubleshooting: - Hangs on DAQ: use `--no-daq` - Protection errors: configure seed/key (see FAQ)

14.3.2 xcp-id-scanner

Scan CAN bus for slaves by broadcasting CONNECT.

Usage:

```
xcp-id-scanner [OPTIONS]
```

Notes: generates bus traffic; limited to CAN; typical range 0x700-0x7FF.

14.3.3 xcp-fetch-a2l

Download A2L from slave (requires `FILE_TO_UPLOAD` and `UPLOAD` support).

Usage:

```
xcp-fetch-a2l [OPTIONS]
```

Output: saves filename reported by slave, or `output.a2l` fallback.

14.3.4 xcp-profile

Create/convert configuration files.

Usage:

```
xcp-profile <create|convert> [OPTIONS]
```

Subcommands: - `create` – generate Python config template - `convert` – migrate legacy JSON/TOML to Python config

14.3.5 xcp-examples

List and copy bundled example scripts.

14.3.6 xcp-discovery

Discover Ethernet slaves via multicast GET_SLAVE_ID / GET_SLAVE_ID_EXTENDED and optionally assign IPv4 by MAC.

Usage:

```
xcp-discovery [OPTIONS]
```

Common options: - `--extended` – send GET_SLAVE_ID_EXTENDED - `--set-ip <MAC> <IP>` – assign an IPv4 to the given MAC - `--dest-address/--dest-port` – override multicast destination - `--response-address/--response-port` – override response group

Example (multicast scan with extended info):

```
xcp-discovery --extended --timeout 5
```

Typical output includes the discovered IP/port, resource flags, and MAC for each responder.

14.3.7 xmraw-converter

Convert recorder `.xmraw` measurement data to CSV/other formats.

14.3.8 pyxcp-probe-can-drivers

List available CAN drivers from python-can backends.

CHAPTER

FIFTEEN

RECORDER

15.1 Overview

pyXCP provides first-class support for XCP DAQ (Data Acquisition) including both online processing and offline recording. This page consolidates guidance and examples from discussions (disc_165) and examples in the repository to help you set up DAQ lists, select an appropriate Policy, record to the native .xmraw format, export CSV in real-time, and post-process recordings into formats like ASAM MDF, Parquet (Arrow), or SQLite.

Key concepts:

- DAQ List: A list of measurements attached to an event on the ECU.
- Policy: A strategy object that drives how DAQ data is handled (online vs offline). You pass the policy into the application context via `ap.run(policy=...)`.
- .xmraw: Proprietary, high-throughput recording format used by pyXCP. Convert later to other formats.
- Timestamps: Two timestamps are supported per DAQ row: `timestamp0` (host-side) and `timestamp1` (device/ECU DAQ timestamp when available in the first ODT). The host timestamp (`timestamp0`) is normally generated by pyXCP; however, you can enable IEEE 1588/PTP hardware timestamps from the NIC by setting the configuration parameter `c.Transport.Eth.ptp_timestamping=True`. If hardware timestamps are not available or not enabled, pyXCP continues to generate the host timestamp.

  Timestamps are normalized, i.e.

  - Both values are in nano-seconds.
  - Both timestamps start with 0.
  - Overflows of the DAQ/ECU timestamp are internally handled (linearization).

15.2 Quick start example (run_daq)

The example `pyxcp/examples/run_daq.py` demonstrates end-to-end DAQ:

```
from pyxcp.cmdline import ArgumentParser
from pyxcp.daq_stim import DaqList, DaqRecorder, DaqToCsv

ap = ArgumentParser(description="DAQ test")

# Define your DAQ lists (addresses below are examples; adjust for your ECU!)
DAQ_LISTS = [
    DaqList(
        name="pwm_stuff",
        event_num=2,
```

(continues on next page)

(continued from previous page)

```
        stim=False,
        enable_timestamps=True,
        measurements=[
            ("channel1", 0x1BD004, 0, "F32"),
            ("period",   0x001C0028, 0, "F32"),
        ],
        priority=0,
        prescaler=1,
    )
]

# Choose a policy:
daq_parser = DaqToCsv(DAQ_LISTS)              # Online CSV per DAQ list
# daq_parser = DaqRecorder(DAQ_LISTS, "run_daq", 2)  # Offline .xmraw recording

with ap.run(policy=daq_parser) as x:
    x.connect()
    x.cond_unlock("DAQ")
    daq_parser.setup()   # allocate and arm DAQs on the slave
    daq_parser.start()   # start acquisition
    import time; time.sleep(60)
    daq_parser.stop()
    x.disconnect()
```

15.3 xcp_daq_recorder Scipt

Usage:

xcp_daq_recorder <daq_config>.json -c <pyxcp_config>.py

Description: This script connects to an XCP slave and records DAQ data according to a JSON configuration file. The configuration uses the "daq_lists" key for the DAQ lists. For backward compatibility, a plain JSON array (without a container object) is also supported.

Expected JSON format (example):

```
{
  "daq_lists": [ ... ],            # list of DAQ lists (or alternatively: JSON root is␣
↪the list)
  "output_type": "xmraw",          # "xmraw" or "csv"
  "output_file": "xcp_rec_0892026",# filename for xmraw output
  "runtime_seconds": 900,          # runtime in seconds (optional)
  "daq_override": { ... }          # optional DAQ processor/resolution override (alias:␣
↪daq_info_override)
}
```

Behavior:

- output_type == "xmraw": a DaqRecorder is created (binary/raw) and output_file is passed as filename.
- output_type == "csv": a DaqToCsv instance is created; output_file is not used, instead CSV files are created per DAQ list with names derived from the DAQ list name.
- runtime_seconds determines the sleep time (time.sleep) during recording. If not provided, a default of 60 seconds is used.

- daq_override / daq_info_override (dict) is passed to `daq_parser.setup(daq_info_override=...)`. Use this when the slave omits optional DAQ services (e.g., GET_DAQ_PROCESSOR_INFO/GET_DAQ_RESOLUTION_INFO). Provide trusted `processor` and `resolution` blocks with `valid` flags if needed; `valid.processor` and `valid.resolution` are forced to `True`.

15.4 Online vs offline and Policies

- Online processing: Use `DaqToCsv` (a `DaqOnlinePolicy`) to process data in-process and write CSV files on-the-fly. This is a convenient starting point for building custom online processing. Another option is `Hdf5OnlinePolicy` which records data to HDF5 files using the `h5py` library.
- Offline recording: Use `DaqRecorder` to record into a high-throughput `.xmraw` file for later, flexible post-processing into multiple formats. This keeps runtime overhead low and allows multiple conversions later.

Pass the chosen policy through `ap.run(policy=...)`. The application wiring ensures incoming DAQ frames are dispatched to the policy.

15.5 Timestamps

If your ECU provides DAQ timestamps in the first ODT and the slave supports timestamps, the DAQ rows include two timestamps:

- `timestamp0`: Host-side, generated by pyXCP, UTC 64-bit with nanosecond resolution.
- `timestamp1`: ECU/device DAQ timestamp (scaled by the slave's timestamp unit and ticks).

Example CSV header:

```
timestamp0,timestamp1,byteCounter,sbyteCounter,wordCounter,dwordCounter
```

15.6 Setting up DAQ lists

Users primarily define two data structures:

- `pyxcp.daq_stim.DaqList`: Corresponds to one XCP DAQ list. Used for dynamic configuration.
- `pyxcp.daq_stim.PredefinedDaqList`: Corresponds to a fixed (predefined) XCP DAQ list on the slave.
- `measurements`: A list of tuples specifying the variables to acquire.

15.6.1 Dynamic DAQ Lists (DaqList)

Conceptually, `DaqList` provides these fields:

```
name: str                # used for naming CSV files, SQL tables, MDF channel groups
event_num: int           # event the DAQ list attaches to
stim: bool               # DAQ (False) or STIM (True); STIM is not implemented yet
enable_timestamps: bool  # whether to include timestamps when selectable
measurements: list       # list[ (name, address, address_extension, datatype) ]
priority: int            # optional
prescaler: int           # optional
```

Allocation and optimization of ODTs is handled automatically by pyXCP (using bin-packing and continuous block construction internally).

15.6.2 Predefined DAQ Lists (PredefinedDaqList)

If the ECU already has a fixed ODT configuration (e.g., for static DAQ setups), use `PredefinedDaqList`. In this case, pyXCP skips `ALLOC_DAQ`, `ALLOC_ODT`, and `WRITE_DAQ` steps and directly starts the acquisition.

```
from pyxcp.daq_stim import PredefinedDaqList

# Define the fixed ODT structure as expected by the slave
odts = [
    [("EngineSpeed", "U16"), ("EngineTemp", "I8")], # ODT 0
    [("ThrottlePos", "U8")]                         # ODT 1
]

daq_list = PredefinedDaqList(
    name="StaticList",
    event_num=1,
    stim=False,
    enable_timestamps=True,
    odts=odts
)
```

Data types are given as mnemonics (e.g., U8, I16, F32, F64, F16, BF16). To see all supported mnemonics:

```
python -c "from pyxcp.recorder import DATA_TYPES; print(DATA_TYPES)"
```

Note about floating point support: - 16-bit floating-point variables are supported where available from the compiler: `F16` (half, 5-bit exponent / 10-bit mantissa) and `BF16` (bfloat16, 8-bit exponent / 7-bit mantissa).

Allocation and optimization of ODTs is handled automatically by pyXCP (using bin-packing and continuous block construction internally).

15.7 Post-processing .xmraw recordings

Use `pyxcp.recorder.XcpLogFileDecoder` as a base class to decode a recorded `.xmraw` file. Hook into `on_daq_list()` to consume rows list-wise per DAQ list.

```
from pathlib import Path
from pyxcp.recorder import XcpLogFileDecoder

class Decoder(XcpLogFileDecoder):
    def __init__(self, recording_file_name: str) -> None:
        self._out = Path(recording_file_name).with_suffix(".txt")

    def initialize(self) -> None:
        self._f = self._out.open("w")

    def finalize(self) -> None:
        self._f.close()

    def on_daq_list(self, daq_list_num: int, timestamp0: int, timestamp1: int,␣
↪measurements: list) -> None:
        self._f.write(f"{timestamp0},{timestamp1},{measurements}\n")

Decoder("my_recording.xmraw").run()
```

15.8 Converters and examples

The repository contains examples demonstrating common conversions of `.xmraw` data:

- `ex_arrow`: Create Apache Parquet files (Arrow).
- `ex_mdf`: Create ASAM MDF files.
- `ex_sqlite`: Create SQLite3 databases.

See `pyxcp/examples` for these scripts. The converter infrastructure lives under `pyxcp/recorder` (see also `recorder/converter` in the source tree if present in your checkout).

15.9 Miscellaneous notes

- Build system: Poetry is used; use `pip install -e .` for editable installs.
- Timestamps are produced by a C++ extension. Startup time (including timezone and DST offsets) is available as `x.start_datetime` within the application context:

```
with ap.run() as x:
    print("Start DT:", x.start_datetime)
```

- Re-using an existing interface: you can pass an existing CAN interface into the `ArgumentParser` context, but ensure your configuration matches the interface type:

```
import can
from pyxcp.cmdline import ArgumentParser

can_if = can.Bus(interface="kvaser", channel="0", fd=False, bitrate=500000)
ap = ArgumentParser(description="external interface test")
with ap.run(transport_layer_interface=can_if) as x:
    x.connect()
    x.disconnect()

can_if.shutdown()  # Ownership is not transferred to pyXCP.
```

  The interface type is currently not deduced, so ensure the configuration matches the passed interface, e.g.:

```
c.Transport.layer = 'CAN'
```

- STIM is not implemented yet, but some infrastructure exists.

CHAPTER

SIXTEEN

PYXCP PACKAGE

16.1 Subpackages

16.1.1 pyxcp.asam package

Submodules

pyxcp.asam.types module

Module contents

16.1.2 pyxcp.master package

Submodules

pyxcp.master.async_master module

pyxcp.master.errorhandler module

pyxcp.master.master module

Module contents

16.1.3 pyxcp.transport package

Submodules

pyxcp.transport.base module

pyxcp.transport.can module

pyxcp.transport.eth module

pyxcp.transport.mock module

pyxcp.transport.sxi module

pyxcp.transport.transport_ext module

pyxcp.transport.usb_transport module

pyxcp.transport.usb_transport_dan202312 module

Module contents

16.2 Submodules

16.3 pyxcp.checksum module

16.4 pyxcp.cmdline module

16.5 pyxcp.constants module

16.6 pyxcp.dllif module

16.7 pyxcp.errormatrix module

16.8 pyxcp.time_correlation module

16.9 pyxcp.time_sync module

16.10 pyxcp.timing module

16.11 pyxcp.types module

16.12 pyxcp.utils module

16.13 Module contents

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search