objutils Documentation

Release 0.10.12

Christoph Schueler

Jul 11, 2026

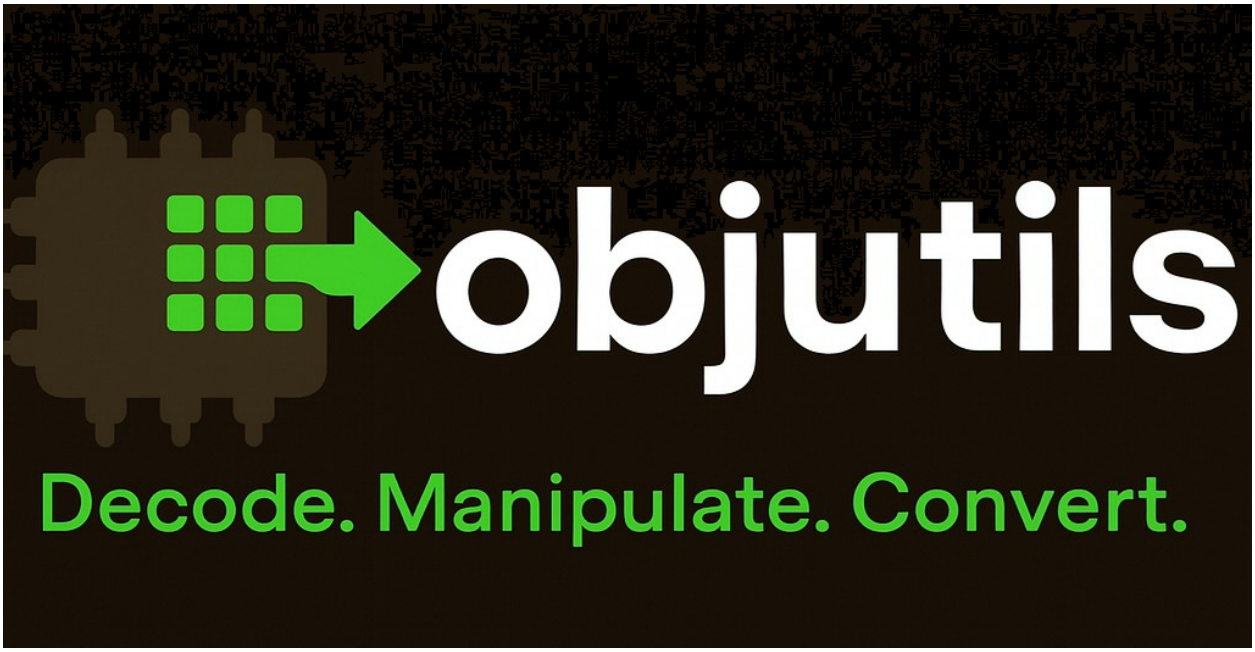
CONTENTS:

objutils provides Python APIs and CLI tools to work with object files and classic HEX/record formats used in embedded firmware workflows. Start with the tutorial and scripts, or jump to the full API reference.

CHAPTER

ONE

README



Binary data stored in hex-files is in widespread use especially in embedded systems applications. `objutils` gives you programmatic access to a wide array of formats and offers a practical API to work with such data.

Get the latest version from Github

1.1 Installation

```
pip install objutils
```

For development (editable install):

```
pip install -e .
```

Or with Poetry:

```
poetry install
```

1.2 Prerequisites

- Python >= 3.9

1.3 Features

- Read ELF files (including symbols) and extract loadable sections.
- Inspect PE/COFF files and symbols (optional PDB support).
- Merge multiple HEX files into one (`oj-hex-merge`).
- Split HEX files into separate sections (`oj-hex-split`).
- Typed access (scalars, arrays, strings) to binary data.

1.4 Recent improvements

- New `oj-hex-merge` and `oj-hex-split` CLI tools for combining and partitioning HEX files.
- New `oj-dwarf-import` CLI replaces `dwarfer.py/cu_info.py` for DWARF imports, CU listing, summaries, and attribute traversal.
- Faster DWARF imports: batched ORM writes, quiet flag propagation, and safe DWARF expression evaluation keep large ELF files stable.
- Exception handling and typing tightened across DWARF/ELF/PECOFF modules and public APIs to surface real errors without masking them.
- `ElfParser.close()` releases SQLite/mmap handles; examples/tests now close parsers to avoid database locks.
- S-Record metadata stays minimal and numeric (no data records mixed in); Mostec/Tek SREC roundtrips now reproduce expected files.
- Fortran-ordered ndarray reads/writes honor legacy column-major expectations; hexdump empty rows keep canonical spacing.

1.4.1 Supported HEX formats

`objutils` supports a bunch of HEX formats…

Current

- codec / format name
- ihex (Intel HEX)
- shf (S Hexdump (rfc4194))
- srec (Motorola S-Records)
- titxt (Texas Instruments Text)

Historical

- codec / format name
- ash (ASCII Space Hex)
- cosmac (RCA Cosmac)
- emon52 (Elektor EMON52)

- etek (Tektronix Extended Hexadecimal)
- fpc (Four Packed Code)
- mostec (MOS Technology)
- rca (RCA)
- sig (Signetics)
- tek (Tektronix Hexadecimal)

codec is the first parameter to dump() / load() functions, e.g.:

```
img = objutils.load("ihex", "myHexFile.hex")     # Load an Intel HEX file...
objutils.dump("srec", "mySRecFile.srec", img)    # and save it as S-Records.
```

1.5 First steps

If you are interested, what `objutils` provides to you out-of-the-box, refer to Scripts documentation.

In any case, you should work through the following tutorial:

First import all classes and functions used in this tutorial.

```
from objutils import Image, Section, dump, dumps, load, loads
```

Everything starts with hello world...

```
sec0 = Section(start_address = 0x1000, data = "Hello HEX world!")
```

The constructor parameters to *Section* reflect what they are about: A continuous area of memory with an start address.

data is not necessarily a string, **array.array**s, **byte**, **bytearray** will also do, or from an internal point of view: everything that is convertible to **bytearray** could be used.

Note: **start_address** and **data** are positional arguments, so there is no need to use them as keywords (just for the sake of illustration).

Now let's inspect our section.

```
sec0.hexdump()

00001000  48 65 6c 6c 6f 20 48 45 58 20 77 6f 72 6c 64 21  |Hello HEX world!|
---------------
       16 bytes
---------------
```

hexdump() gives us, what in the world of hackers is known as a canonical hexdump.

HEX files usually consist of more than one section, so let's create another one.

```
sec1 = Section(0x2000, range(1, 17))
sec1.hexdump()

00002000  01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10  |................|
---------------
       16 bytes
---------------
```

Now, let's glue together our sections.

```
img0 = Image([sec0, sec1])
print(img0)

Section(address = 0X00001000, length = 16, data = b'Hello HEX world!')
Section(address = 0X00002000, length = 16, data = b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\
↪x0b\x0c\r\x0e\x0f\x10')
```

Images are obviously a container for sections, and they are always involved if you are interacting with disk based HEX files.

```
dump("srec", "example0.srec", img0)
```

The resulting file could be inspected from command line.

```
$ cat example0.srec
S113100048656C6C6F2048455820776F726C64217A
S11320000102030405060708090A0B0C0D0E0F1044
```

And loaded again…

```
img1 = load("srec", "example0.srec")
print(img1)

Section(address = 0X00001000, length = 16, data = b'Hello HEX world!')
Section(address = 0X00002000, length = 16, data = b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\
↪x0b\x0c\r\x0e\x0f\x10')
```

Or split it into its sections…

```
images = img1.split()
for i, img in enumerate(images):
    dump("srec", f"part_{i}.srec", img)
```

This leads to the conversion idiom.

```
img1 = load("srec", "example0.srec")
dump("ihex", "example0.hex", img1)
```

Note: the formats above listed as historical are for one good reason historical: they are only 16bit wide, so if you want to convert, say a **srec** file for a 32bit MCU to them, you're out of luck.

OK, we're starting another session.

```
sec0 = Section(0x100, range(1, 9))
sec1 = Section(0x108, range(9, 17))
img0 = Image([sec0, sec1])
print(img0)

Section(address = 0X00000100, length = 16, data = b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\
↪x0b\x0c\r\x0e\x0f\x10')

img0.hexdump()
```

(continues on next page)

(continued from previous page)

```
Section #0000
-------------
00000100  01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10  |................|
---------------
       16 bytes
---------------
```

Two sections with consecutive address ranges concatenated to one, this may or may not what you are expected.

For this reason **Image** has a **join** parameter.

```
sec0 = Section(0x100, range(1, 9))
sec1 = Section(0x108, range(9, 17))
img0 = Image([sec0, sec1], join = False)
print(img0)

Section(address = 0X00000100, length = 8, data = b'\x01\x02\x03\x04\x05\x06\x07\x08')
Section(address = 0X00000108, length = 8, data = b'\t\n\x0b\x0c\r\x0e\x0f\x10')

img0.hexdump()

Section #0000
-------------
00000100  01 02 03 04 05 06 07 08                           |........        |
---------------
        8 bytes
---------------

Section #0001
-------------
00000108  09 0a 0b 0c 0d 0e 0f 10                           |........        |
---------------
        8 bytes
---------------
```

One feature that sets **objutils** apart from other libraries of this breed is typified access.

We are starting with a new image.

```
img0 = Image([Section(0x1000, bytes(64))])
print(img0)

Section(address = 0X00001000, length = 64, data = b'\x00\x00\x00\x00\x00\x00\x00...00\
↪x00\x00\x00\x00\x00\x00\x00')
```

We are now writing a string to our image.

```
img0 = Image([Section(0x1000, bytes(64))])
img0.write(0x1010, [0xff])
img0.hexdump()

Section #0000
-------------
00001000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

(continues on next page)

(continued from previous page)

```
00001010  ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00001020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|

img0.write_string(0x1000, "Hello HEX world!")
img0.hexdump()

Section #0000
-------------
00001000  48 65 6c 6c 6f 20 48 45 58 20 77 6f 72 6c 64 21  |Hello HEX world!|
00001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
          *
00001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
---------------
       64 bytes
---------------
```

Notice the difference? In our **Section** example above, the string passed as a **data** parameter was just a bunch of bytes, but now it is a "real" C-string (there is a opposite function, **read_string**, that scans for a terminating **NULL** character).

Use **write()** and **read()** functions, if you want to access plain bytes.

But there is also support for numerical types.

```
img0 = Image([Section(0x1000, bytes(64))])
img0.write_numeric(0x1000, 0x10203040, "uint32_be")
img0.write_numeric(0x1004, 0x50607080, "uint32_le")
img0.hexdump()

Section #0000
-------------
00001000  10 20 30 40 80 70 60 50 00 00 00 00 00 00 00 00  |. 0@.p`P........|
00001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
          *
00001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
---------------
       64 bytes
---------------
```

The folling types are supported:

- uint8
- int8
- uint16
- int16
- uint32
- int32
- uint64
- int64
- float32

- float64

In any case, endianess suffixes **_be** or **_le** are required.

For ASAM workflows there are dedicated helpers with explicit byte-order names:

```
img0 = Image([Section(0x1000, bytes(64))])
img0.write_asam_numeric(0x1000, 0x11223344, "ULONG", "MSB_FIRST")
img0.write_asam_numeric(0x1004, 0x11223344, "ULONG", "MSB_FIRST_MSW_LAST")
img0.write_asam_numeric(0x1008, 0x11223344, "ULONG", "MSB_LAST_MSW_FIRST")

print(hex(img0.read_asam_numeric(0x1000, "ULONG", "MSB_FIRST")))
print(hex(img0.read_asam_numeric(0x1004, "ULONG", "MSB_FIRST_MSW_LAST")))
print(hex(img0.read_asam_numeric(0x1008, "ULONG", "MSB_LAST_MSW_FIRST")))

# All reads print 0x11223344 again.
```

Supported ASAM byte orders:

- MSB_FIRST
- MSB_LAST
- MSB_FIRST_MSW_LAST (word-swap)
- MSB_LAST_MSW_FIRST (word-swap)
- LITTLE_ENDIAN (legacy alias for MSB_LAST)
- BIG_ENDIAN (legacy alias for MSB_FIRST)

Supported ASAM numeric datatypes:

- UBYTE, SBYTE
- UWORD, SWORD
- ULONG, SLONG
- A_UINT64, A_INT64
- FLOAT16_IEEE, FLOAT32_IEEE, FLOAT64_IEEE

ASAM string helpers are available, too:

```
img0.write_asam_string(0x1020, "MOTOR", "ASCII")
img0.write_asam_string(0x1030, "Drehzahl", "UTF8")

print(img0.read_asam_string(0x1020, "ASCII"))
print(img0.read_asam_string(0x1030, "UTF8"))
```

Supported ASAM string datatypes:

- ASCII
- UTF8
- UTF16
- UTF32

Arrays are also supported.

```
img0 = Image([Section(0x1000, bytes(64))])
img0.write_numeric_array(0x1000, [0x1000, 0x2000, 0x3000, 0x4000, 0x5000, 0x6000, 0x7000,
↪ 0x8000], "uint16_le")
img0.hexdump()

Section #0000
-------------
00001000  00 10 00 20 00 30 00 40 00 50 00 60 00 70 00 80  |... .0.@.P.`.p..|
00001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
          *
00001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
---------------
       64 bytes
---------------
```

This concludes our tutorial for now, but there is more stuff to follow...

1.6 Documentation

For full documentation, including installation, tutorials and PDF documents, please see Readthedocs

1.7 Bugs/Requests

Please use the GitHub issue tracker to submit bugs or request features

1.8 References

Here is an overview of some of the classic hex-file formats.

1.9 Authors

- Christoph Schueler - Initial work and project lead.

1.10 License

This project is licensed under the GNU General Public License v2.0

1.11 Contribution

If you contribute code to this project, you are implicitly allowing your code to be distributed under the GNU General Public License v2.0. You are also implicitly verifying that all code is your original work.

CHAPTER

TWO

INSTALLATION AND GETTING STARTED

2.1 Supported Python and platforms

- Python >= 3.9 (CPython; PyPy not tested)
- Cross-platform; no OS-specific restrictions

2.2 Install

From PyPI:

```
pip install objutils
```

For development (editable install):

```
pip install -e .
```

Or with Poetry:

```
poetry install
```

2.3 Quick verification

Open a Python REPL and run:

```
from objutils import Image, Section, dump
img = Image([Section(0x1000, b"hi")])
dump("srec", "verify.srec", img)
print("Wrote verify.srec")
```

2.4 Next steps

- Read the Tutorial for a guided introduction.
- See HOW-TOs for task-oriented examples.
- Check Scripts for CLI utilities.
- Full docs: latest on Read the Docs.

CHAPTER

THREE

TUTORIAL

This tutorial walks you through the basics of working with objutils: creating sections and images, converting between HEX formats, controlling join behavior, and using typed access helpers.

If you prefer runnable examples, see the scripts and examples in the repository.

3.1 Before you start

- Install the package: `pip install objutils`
- Basic familiarity with Python byte sequences

3.2 Hello, HEX world

Start by importing the primary entry points:

```
from objutils import Image, Section, dump, load, dumps, loads
```

Create two sections and inspect them:

```
sec0 = Section(start_address=0x1000, data=b"Hello HEX world!")
sec1 = Section(0x2000, range(1, 17))

img = Image([sec0, sec1])
img.hexdump()
```

Persist as S-Records and read back as Intel HEX:

```
dump("srec", "example.srec", img)
img2 = load("srec", "example.srec")
dump("ihex", "example.hex", img2)
```

3.3 Join vs. no-join

By default, consecutive sections are joined into a single section when possible. You can disable this:

```
s0 = Section(0x100, range(1, 9))
s1 = Section(0x108, range(9, 17))

img_joined = Image([s0, s1])           # default join=True
img_nojoin = Image([s0, s1], join=False)
```

(continues on next page)

(continued from previous page)

```
img_joined.hexdump()
img_nojoin.hexdump()
```

3.4 Typed access (strings, numbers, arrays)

Use the typed helpers to read/write structured binary data with explicit endianness.

```
img = Image([Section(0x1000, bytes(64))])

# Strings (C-style NUL-terminated)
img.write_string(0x1000, "Hello HEX world!")

# Scalars with endianness
img.write_numeric(0x1010, 0x10203040, "uint32_be")
img.write_numeric(0x1014, 0x50607080, "uint32_le")

# Arrays
img.write_numeric_array(0x1018, [0x1000, 0x2000, 0x3000], "uint16_le")

img.hexdump()
```

Supported scalar types:

- uint8, int8
- uint16, int16
- uint32, int32
- uint64, int64
- float32, float64

An endianness suffix (`_be` or `_le`) is required.

3.5 ASAM byte order and datatype helpers

For ECU/ASAM style type names and byte orders (including word-swap variants), use the dedicated ASAM helpers:

Use these helpers when your calibration metadata uses ASAM type names (`ULONG`, `UWORD`, `FLOAT32_IEEE`) and ECU byte-order terms (`MSB_FIRST`, `MSB_LAST_MSW_FIRST`).

When to use ASAM helpers instead of plain `read_numeric*`/`write_numeric*`:

- Your metadata comes from A2L/ASAM naming (for example `UWORD`/`ULONG`).
- You need ECU-specific byte order terms and MSW swapping.
- You want one consistent API for scalars, Python lists, and NumPy arrays.

For signatures and parameter semantics of the array helpers, see the `ASAM Array Cheat Sheet` below.

The example below shows ASAM scalars, Python arrays, NumPy arrays, and strings side by side in one small image.

```
from objutils import Image, Section
import numpy as np

img = Image([Section(0x3000, bytes(96))])

# ASAM numerics
img.write_asam_numeric(0x3000, 0x11223344, "ULONG", "MSB_FIRST")
img.write_asam_numeric(0x3004, 0x11223344, "ULONG", "MSB_FIRST_MSW_LAST")
img.write_asam_numeric(0x3008, 0x11223344, "ULONG", "MSB_LAST_MSW_FIRST")

# Roundtrip reads
a = img.read_asam_numeric(0x3000, "ULONG", "MSB_FIRST")
b = img.read_asam_numeric(0x3004, "ULONG", "MSB_FIRST_MSW_LAST")
c = img.read_asam_numeric(0x3008, "ULONG", "MSB_LAST_MSW_FIRST")

# ASAM numeric arrays
img.write_asam_numeric_array(0x3020, [0x11223344, 0x55667788], "ULONG", "MSB_LAST_MSW_
↪FIRST")
arr_values = img.read_asam_numeric_array(0x3020, 2, "ULONG", "MSB_LAST_MSW_FIRST")

# ASAM ndarrays
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.uint16)
img.write_asam_ndarray(0x3040, arr, "UWORD", "MSB_FIRST", index_mode="COLUMN_DIR")
arr_roundtrip = img.read_asam_ndarray(0x3040, 6, "UWORD", shape=(3, 2), index_mode=
↪"COLUMN_DIR", byte_order="MSB_FIRST")

# ASAM strings
img.write_asam_string(0x3010, "MOTOR", "ASCII")
img.write_asam_string(0x3030, "Drehzahl", "UTF8")
s0 = img.read_asam_string(0x3010, "ASCII")
s1 = img.read_asam_string(0x3030, "UTF8")
```

NumPy ASAM roundtrip with matrix data

```
from objutils import Image, Section
import numpy as np

img = Image([Section(0x5000, bytes(64))])

matrix = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.uint16)
img.write_asam_ndarray(0x5000, matrix, "UWORD", "MSB_FIRST", index_mode="COLUMN_DIR")

matrix_rt = img.read_asam_ndarray(0x5000, 6, "UWORD", shape=(3, 2), index_mode="COLUMN_
↪DIR", byte_order="MSB_FIRST")
assert np.array_equal(matrix_rt, matrix)
```

Supported ASAM byte orders

- `MSB_FIRST` (big-endian)
- `MSB_LAST` (little-endian)
- `MSB_FIRST_MSW_LAST` (word-swapped)

- `MSB_LAST_MSW_FIRST` (word-swapped)
- `LITTLE_ENDIAN` (legacy alias for `MSB_LAST`)
- `BIG_ENDIAN` (legacy alias for `MSB_FIRST`)

Supported ASAM numeric datatypes

- `UBYTE`, `SBYTE`
- `UWORD`, `SWORD`
- `ULONG`, `SLONG`
- `A_UINT64`, `A_INT64`
- `FLOAT16_IEEE`, `FLOAT32_IEEE`, `FLOAT64_IEEE`

Supported ASAM string datatypes

- `ASCII`
- `UTF8`
- `UTF16`
- `UTF32`

ASAM Array Cheat Sheet

Quick reference for the ASAM array helpers on `Image` and `Section`.

Method	`length` semantics	Returns	Typical usage
`read_asam_numeric_array(...)`	element count	`tuple[int]` / `tuple[float]`	scalar lists/tuples
`write_asam_numeric_array(...)`	n/a (from `len(data)`)	`None`	scalar lists/tuples
`read_asam_ndarray(...)`	element count	`numpy.ndarray`	matrix/tensor data
`write_asam_ndarray(...)`	n/a (from `array.nbytes`)	`None`	matrix/tensor data

Minimal signatures

- `read_asam_numeric_array(addr, length, dtype, byte_order="MSB_LAST")`
- `write_asam_numeric_array(addr, data, dtype, byte_order="MSB_LAST")`
- `read_asam_ndarray(addr, length, dtype, shape=None, byte_order="MSB_LAST", index_mode="ROW_DIR")`
- `write_asam_ndarray(addr, array, dtype, byte_order="MSB_LAST", index_mode="ROW_DIR")`

Note

ASAM index modes

`index_mode="ROW_DIR"` (default) uses C-like row-major layout where X increments fastest. `index_mode="COLUMN_DIR"` swaps only X and Y (not true Fortran-order for dims > 2).

`shape` uses **ASAM convention** `(X, Y, Z, ...)` which is reversed compared to numpy `(..., Z, Y, X)`. `length` is the **element count**, not byte count.

Warning

Frequent pitfalls

- Confusing element count vs. byte count for `length`.
- Forgetting that byte order is applied per element, not per full buffer.
- Passing unsupported ASAM dtype names (must be values like `UWORD`/`ULONG`).
- Assuming MSW swapping affects 8-bit types (it does not).

Copy/paste example: ULONG array roundtrip

```
from objutils import Image, Section

img = Image([Section(0x6000, bytes(32))])

img.write_asam_numeric_array(0x6000, [0x11223344, 0x55667788], "ULONG", "MSB_LAST_MSW_
↪FIRST")

# Optional: verify raw in-memory bytes.
assert img.read(0x6000, 8) == b"\x33\x44\x11\x22\x77\x88\x55\x66"

# Main check: logical values roundtrip correctly.
values = img.read_asam_numeric_array(0x6000, 2, "ULONG", "MSB_LAST_MSW_FIRST")
assert values == (0x11223344, 0x55667788)
```

3.6 CLI companions

The library ships with handy command-line tools. A few favorites:

- `oj-hex-info`: inspect HEX files, optionally with a hexdump (`-d`)
- `oj-elf-extract`: extract loadable sections from an ELF to HEX (ihex/shf/srec)
- `oj-elf-arm-attrs`: dump `.ARM.attributes` from an ELF

Examples:

```
oj-hex-info srec example.srec -d
oj-elf-extract build/app.elf app.srec -t srec
```

3.7 What next?

- See HOW-TOs for short, task-oriented recipes.
- Explore the full API reference in the modules section.

CHAPTER

FOUR

HOW-TOS

Task-oriented examples and small recipes for common workflows.

4.1 Convert between HEX formats (API)

```
from objutils import load, dump
img = load("ihex", "in.hex")
dump("srec", "out.srec", img)
```

4.2 Convert between HEX formats (CLI)

```
# Inspect input
oj-hex-info ihex in.hex

# Convert via a temporary Image in Python (see API example above), or use
# your own small script to wire load/dump in CI.
```

4.3 Join or keep separate sections

By default, adjacent/overlapping sections may be merged. Disable joining to keep boundaries:

```
from objutils import Image, Section
img = Image([Section(0x100, range(8)), Section(0x108, range(8))], join=False)
img.hexdump()
```

4.4 Pretty hexdumps for reviews

```
from objutils import Image, Section
Image([Section(0x1000, b"example")]).hexdump()
```

4.5 Read/write typed values at absolute addresses

```
from objutils import Image, Section
img = Image([Section(0x2000, bytes(32))])
img.write_numeric(0x2000, 0x12345678, "uint32_be")
```

(continues on next page)

(continued from previous page)

```
img.write_numeric_array(0x2004, [1, 2, 3, 4], "uint16_le")
img.write_string(0x2010, "hello")
```

4.6 Read/write ASAM values (incl. word-swap byte orders)

```
from objutils import Image, Section

img = Image([Section(0x3000, bytes(64))])

# ASAM numeric helpers
img.write_asam_numeric(0x3000, 0x11223344, "ULONG", "MSB_FIRST")
img.write_asam_numeric(0x3004, 0x11223344, "ULONG", "MSB_FIRST_MSW_LAST")
img.write_asam_numeric(0x3008, 0x11223344, "ULONG", "MSB_LAST_MSW_FIRST")

value0 = img.read_asam_numeric(0x3000, "ULONG", "MSB_FIRST")
value1 = img.read_asam_numeric(0x3004, "ULONG", "MSB_FIRST_MSW_LAST")
value2 = img.read_asam_numeric(0x3008, "ULONG", "MSB_LAST_MSW_FIRST")

# ASAM string helpers
img.write_asam_string(0x3010, "MOTOR", "ASCII")
name = img.read_asam_string(0x3010, "ASCII")
```

4.7 Extract loadable image from ELF

Use the CLI to generate HEX for flashing:

```
oj-elf-extract build/app.elf app.srec -t srec
```

4.8 Extract loadable image from PE/COFF (32-bit and 64-bit)

For 32-bit PE files the default behaviour works out of the box:

```
oj-coff-extract app32.exe app32.hex -t ihex
```

64-bit PE files typically have an image base of `0x140000000` or higher. When the image base is added to section RVAs, the resulting absolute addresses exceed the 32-bit limit (`0xFFFFFFFF`) that Intel HEX and Motorola S-Record formats can represent. The tool will abort with an *"address too large"* error in that case.

Use the `--no-image-base` (`-r`) flag to emit **relative virtual addresses** (RVAs) instead. RVAs start at zero and therefore stay well within 32-bit range:

```
# Will fail for a typical 64-bit PE (image base 0x140000000)
oj-coff-extract app64.exe app64.hex

# Use --no-image-base to subtract the image base
oj-coff-extract app64.exe app64.hex --no-image-base
```

The tool prints which mode is active so you can verify:

```
Using relative addresses (image base 0x140000000 subtracted).
```

> **Note**
>
> When `--no-image-base` is used, the addresses in the output file are offsets from the PE image base. Your flash-programming tool or linker script must account for this by adding the base back at load time.

4.9 Inspect HEX files

```
# Show section addresses and lengths only
oj-hex-info srec app.srec

# Include a hexdump of sections
oj-hex-info srec app.srec -d
```

4.10 Where to go next

- See the Tutorial for a guided walk-through.
- Refer to Scripts for comprehensive CLI usage and options.

CHAPTER

FIVE

SCRIPTS PROVIDED BY OBJUTILS

`objutils` contains some more or less useful scripts…

5.1 oj_elf_arm_attrs

```
usage: oj-elf-arm-attrs [-h] elf_file

Dump '.ARM.attributes' section.

positional arguments:
  elf_file    .elf file

  optional arguments:
    -h, --help  show this help message and exit
```

You may run the following on your RaspberryPI:

```
$ oj-elf-arm-attrs /usr/bin/gcc


=====
aeabi
=====

Name                            Value
Description
-------------------------------------------------------------------------------
Tag_CPU_name                    6
6

Tag_CPU_arch                    6
ARM v6

Tag_ARM_ISA_use                 1
The user intended that this entity could use ARM instructions

Tag_THUMB_ISA_use               1
The user permitted this entity to use 16-bit Thumb instructions (including BL)

Tag_FP_arch                     2
Use of the v2 FP ISA was permitted (implies use of the v1 FP ISA)
```

(continues on next page)

(continued from previous page)

```
Tag_ABI_PCS_wchar_t             4
The user intended the size of wchar_t to be 4

Tag_ABI_FP_rounding             1
The user permitted this code to choose the IEEE 754 rounding mode at run time

Tag_ABI_FP_denormal             1
The user permitted this code to choose the IEEE 754 rounding mode at run time

Tag_ABI_FP_exceptions           1
The user permitted this code to check the IEEE 754 inexact exception

Tag_ABI_FP_number_model         3
The user permitted this code to use all the IEEE 754-defined FP encodings

Tag_ABI_align_needed            1
Code was permitted to depend on the 8-byte alignment of 8-byte data items

Tag_ABI_align8_preserved        1
Code was required to preserve 8-byte alignment of 8-byte data objects

Tag_ABI_enum_size               2
The user intended Enum containers to be 32-bit

Tag_ABI_VFP_args                1
The user intended FP parameter/result passing to conform to AAPCS, VFP variant

Tag_CPU_unaligned_access        1
The user intended that this entity might make v6-style unaligned data accesses
```

5.2 oj_elf_extract

Extract sections from *ELF* suitible for flashing;

```
Extract sections contributing to program image, e.g. for flash programming
applications.

positional arguments:
  elf_file              ELF file
  output_file_name      Output filename.

optional arguments:
  -h, --help            show this help message and exit
  -j, --join            Try to make continuous sections (merge adjacent ranges)
  -t {ihex,shf,srec,titxt}, --file-type {ihex,shf,srec,titxt}
                        Type of output HEX file (default: ihex)
  -e EXCLUDE, --exclude_pattern EXCLUDE
                        Exclude sections matching a Python regex
  -i INCLUDE, --include_pattern INCLUDE
                        Include only sections matching a Python regex
```

For example:

```
$ oj-elf-extract sample_proj.elf sample_proj.srec -t srec

Extracting from...

Section                   Address    Length
-----------------------------------------------
.text                     0x00000000 46652
.rodata                   0x0000b640 2328
.data                     0x40002000 1996
.sdata                    0x400027cc 16
.eh_frame                 0x400027dc 92
-----------------------------------------------
HEX image written to: 'sample_proj.srec' [51084 total bytes]
```

5.3 oj_elf_import

Import DWARF sections from an ELF into a `.prgdb` SQLite database.

```
oj-elf-import path\to\program.elf
oj-elf-import path\to\program.elf --out program.prgdb --force
```

5.4 oj_dwarf_import

Import DWARF sections into a `.prgdb` database. Use `oj-dwarf-info` for read-only inspection of existing databases.

```
usage: oj-dwarf-import [-h] [--out-db OUT_DB] [--force] [--quiet] [--verbose]
                       [--skip-lines] [--skip-pubnames] [--skip-aranges] [--skip-mac]
                       elf

positional arguments:
  elf                   ELF file with DWARF sections

optional arguments:
  --out-db OUT_DB       Output .prgdb path (default: <elf>.prgdb)
  --force               Overwrite existing database
  --quiet, -q           Suppress non-error output
  --verbose, -v         Verbose DWARF processing output
  --skip-lines          Skip .debug_line processing
  --skip-pubnames       Skip .debug_pubnames processing
  --skip-aranges        Skip .debug_aranges processing
  --skip-mac            Skip .debug_macro processing
```

Examples:

```
oj-dwarf-import program.elf --out-db program.prgdb
oj-dwarf-import program.elf --force --quiet
```

5.5 oj_dwarf_info

Inspect DWARF data from an existing `.prgdb` database without performing an import.

```
usage: oj-dwarf-info [-h] [--quiet] [--list-cus] [--summary]
                     [--walk-attrs] [--offset OFFSET]
                     db

positional arguments:
  db                    Path to .prgdb database generated by oj-dwarf-import

optional arguments:
  --quiet, -q           Suppress non-error output
  --list-cus            List compile units
  --summary             Print DIE/attribute counts
  --walk-attrs          Traverse DIE attributes (starts at first DIE unless --offset is␣
↪set)
  --offset OFFSET       Absolute DIE offset (decimal or 0x-prefixed hex) used for --walk-
↪attrs
```

Examples:

```
oj-dwarf-info program.prgdb --summary --list-cus
oj-dwarf-info program.prgdb --walk-attrs --offset 0x1234 -q
```

5.6 oj_cgen

Generate C/C++ declarations from a `.prgdb` database using DWARF DIEs.

```
oj-cgen program.prgdb --out generated.h
oj-cgen program.prgdb --start 0x14182 --no-guard
```

5.7 oj_elf_info

```
usage: oj-elf-info [-h] [-k] [-l LOGLEVEL] [-S] [-u] elf_file

Display informations about ELF files.

positional arguments:
  elf_file              ELF file

optional arguments:
  -h, --help            show this help message and exit
  -k                    keep directory; otherwise create db in current directory
  -l LOGLEVEL           loglevel [warn | info | error | debug]
  -S, --sections, --section-headers
                        Display the sections' headers.
  -u                    Generate UTF-8 encoded output (otherwise Latin-1).
```

Example:

```
oj-elf-info build/app.elf
```

This prints ELF class, type, machine, byte-order, OS/ABI, followed by a sections table and common notes/comments when present.

5.8 oj_elf_syms

```
usage: oj-elf-syms [-h] [-s SECTIONS] [-p PATTERN] [-t TYPES]
                   [-a ACCESS] [-b BINDINGS] [-o {N,V}] elf_file

Display ELF symbols.

positional arguments:
  elf_file              ELF file

optional arguments:
  -h, --help            show this help message and exit
  -s SECTIONS, --sections SECTIONS
                        Use only symbols from listed sections (comma-separated)
  -p PATTERN, --pattern PATTERN
                        Only display symbols matching a (Python) regex
  -t TYPES, --types TYPES
                        Use only symbols with listed types (comma-separated)
  -a ACCESS, --access ACCESS
                        Filter by access flags: A (allocate), W (write), X (execute)
  -b BINDINGS, --bindings BINDINGS
                        Use only symbols with listed bindings (comma-separated)
  -o {N,V}, --order-by {N,V}
                        Order symbols by Name or Value (default: V)
```

Examples:

```
# All symbols ordered by address
oj-elf-syms build/app.elf -o V

# Only functions from .text, ordered by name
oj-elf-syms build/app.elf -s .text -t FUNC -o N

# Filter by regex and show only GLOBAL bindings that are executable
oj-elf-syms build/app.elf -p '^(reset|_?start)$' -b GLOBAL -a X
```

5.9 oj_hex_info

```
usage: oj-hex-info [-h] [-d] [-p]
                   [file_type] hex_file

Displays informations about HEX files.

positional arguments:
  file_type             file type (optional, if omitted it is probed)
  hex_file              HEX file
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -d, --dump            hexdump contents
  -p, --print-filename  Print filename including path
```

Run it as follows:

Without any optional arguments just the addresses and lengths of the contained sections are shown:

```
$ oj-hex-info sample.srec

Sections
--------

Num   Address    Length
-------------------------------------------------------------------------------
000   0x00001000 16
001   0x00002000 16
-------------------------------------------------------------------------------
                 32 total bytes
```

If you also want to see the contents, add *-d* option:

```
$ oj-hex-info sample.srec -d

Sections
--------

Num   Address    Length
-------------------------------------------------------------------------------
000   0x00001000 16
001   0x00002000 16
-------------------------------------------------------------------------------
                 32 total bytes

Section #0000
-------------
00001000  48 65 6c 6c 6f 20 48 45 58 20 77 6f 72 6c 64 21  |Hello HEX world!|
---------------
       16 bytes
---------------

Section #0001
-------------
00002000  01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10  |................|
---------------
       16 bytes
---------------
```

5.10 oj_hex_merge

```
usage: oj-hex-merge [-h] -o OUTPUT [-t OUTPUT_TYPE] [-p] [-v]
                    input_files [input_files ...]

Merges multiple HEX files into one.

positional arguments:
  input_files           Input HEX files to be merged.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output HEX file.
  -t OUTPUT_TYPE, --type OUTPUT_TYPE
                        Output file type (e.g., ihex, srec). If omitted, it's
                        inferred from the output file extension.
  -p, --print-filename  Print filenames while processing
  -v, --verbose         Verbose output
```

Example:

```
$ oj-hex-merge file1.hex file2.hex -o merged.hex
```

Note: Merging files with overlapping sections is not allowed and will result in an error.

5.11 oj_hex_split

```
usage: oj-hex-split [-h] [-o OUTPUT_NAMES [OUTPUT_NAMES ...]]
                    [-t OUTPUT_TYPE] [-p PREFIX] [-v]
                    input_file

Splits a HEX file into multiple files based on its sections.

positional arguments:
  input_file            Input HEX file to be split.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_NAMES [OUTPUT_NAMES ...], --output-names OUTPUT_NAMES [OUTPUT_NAMES ...]
                        Optional list of output base names. If not provided,
                        section numbers are used.
  -t OUTPUT_TYPE, --type OUTPUT_TYPE
                        Output file type (e.g., ihex, srec). If omitted, it's
                        inferred from the input file extension.
  -p PREFIX, --prefix PREFIX
                        Prefix for output filenames. Defaults to input
                        filename (without extension).
  -v, --verbose         Verbose output
```

Example:

```
$ oj-hex-split merged.hex
```

This will generate files like `merged_000.hex`, `merged_001.hex`, etc.

You can also specify custom names:

```
$ oj-hex-split merged.hex -o code data
```

5.12 oj_coff_info

Display information about PE/COFF files (EXE/DLL/OBJ), optionally with PDB symbols.

```
oj-coff-info app.exe
oj-coff-info app.exe --pdb app.pdb
```

5.13 oj_coff_syms

List symbols contained in a PE/COFF file (COFF table or PDB if available).

```
oj-coff-syms app.exe
oj-coff-syms app.exe --pattern printf --order-by N
```

5.14 oj_coff_extract

Extract sections contributing to the program image from a PE/COFF file.

```
usage: oj-coff-extract [-h] [-j] [-t {ihex,shf,srec,titxt}]
                       [-e EXCLUDE] [-i INCLUDE] [-r] [-n ROW_LENGTH]
                       pe_file output_file_name

positional arguments:
  pe_file               PE/COFF file
  output_file_name      Output filename.

optional arguments:
  -h, --help            show this help message and exit
  -j, --join            Try to make continuous sections.
  -t {ihex,shf,srec,titxt}, --file-type {ihex,shf,srec,titxt}
                        Type of output HEX file (default: ihex)
  -e EXCLUDE, --exclude_pattern EXCLUDE
                        Exclude sections containing this substring
  -i INCLUDE, --include_pattern INCLUDE
                        Include only sections containing this substring
  -r, --no-image-base   Use relative virtual addresses (RVAs) instead of
                        absolute addresses. Required for 64-bit PE files
                        whose image base pushes addresses beyond 32-bit
                        hex format limits.
  -n ROW_LENGTH         Number of data bytes per line (default: 16)
```

Examples:

```
oj-coff-extract app.exe app.srec -t srec
oj-coff-extract app.exe app.hex -t ihex -j

# 64-bit PE: use relative addresses to stay within 32-bit hex limits
oj-coff-extract app64.exe app64.hex --no-image-base
```

5.15 oj_coff_import

Import PE/COFF header, sections, and symbols into a `.pedb` SQLite database.

```
oj-coff-import app.exe
oj-coff-import app.exe --out app.pedb --force
```

5.16 arduino_build_artifacts

Locate Arduino build artifacts (ELF/HEX/EEP/MAP) produced by the Arduino IDE for a given sketch. You can pass either the path to a .ino file or the sketch directory.

```
usage: arduino-build-artifacts [-h] [--only {DIRECTORY,ELF,HEX,EEP,MAP} ...]
                               [--as-paths] [--missing-ok] [--quiet]
                               sketch

positional arguments:
  sketch                Path to the sketch directory or .ino file

options:
  -h, --help            show this help message and exit
  --only {DIRECTORY,ELF,HEX,EEP,MAP} ...
                        Limit output to specific artifact types (may be given multiple␣
↪times)
  --as-paths            Print only paths (one per line) without labels
  --missing-ok          Do not treat missing artifacts as an error; just omit them from␣
↪output
  -q, --quiet           Suppress non-error output
```

Examples:

```
# Show all available artifacts with labels
$ arduino-build-artifacts MySketch/MySketch.ino
DIRECTORY: C:\\Users\\<you>\\AppData\\Local\\arduino\\sketches\\ABCD1234...
ELF      : C:\\...\\MySketch.ino.elf
HEX      : C:\\...\\MySketch.ino.hex
MAP      : C:\\...\\MySketch.ino.map

# Print only the ELF and HEX paths, one per line
$ arduino-build-artifacts MySketch --only ELF HEX --as-paths
C:\\...\\MySketch.ino.elf
C:\\...\\MySketch.ino.hex

# Ignore missing artifacts (e.g., no EEP generated)
$ arduino-build-artifacts MySketch --only ELF EEP --missing-ok
```

CHAPTER

SIX

INTERACTIVE SESSION

A guided REPL walkthrough using the current API. All snippets can be pasted directly into a Python interpreter.

6.1 Imports

```
>>> from objutils import Image, Section, dump, dumps, load, loads
```

6.2 Hello, HEX world

Create a section and inspect it:

```
>>> sec0 = Section(start_address=0x1000, data=b"Hello HEX world!")
>>> sec0.hexdump()
00001000  48 65 6c 6c 6f 20 48 45 58 20 77 6f 72 6c 64 21  |Hello HEX world!|
---------------
       16 bytes
---------------
```

A second section with byte values 1–16:

```
>>> sec1 = Section(0x2000, range(1, 17))
>>> sec1.hexdump()
00002000  01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10  |................|
---------------
       16 bytes
---------------
```

Combine sections into an image and print it:

```
>>> img = Image([sec0, sec1])
>>> print(img)
Section(address = 0X00001000, length = 16, data = b'Hello HEX world!')
Section(address = 0X00002000, length = 16, data = b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\
↪x0b\x0c\r\x0e\x0f\x10')
```

6.3 Format conversion

Serialize to Motorola S-Records and read back:

```
>>> dump("srec", "example.srec", img)
>>> img2 = load("srec", "example.srec")
>>> print(img2)
Section(address = 0X00001000, length = 16, data = b'Hello HEX world!')
Section(address = 0X00002000, length = 16, data = b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\
↪x0b\x0c\r\x0e\x0f\x10')
```

Convert to Intel HEX:

```
>>> dump("ihex", "example.hex", img2)
```

In-memory round-trip with `dumps`/`loads`:

```
>>> raw = dumps("srec", img)
>>> img3 = loads("srec", raw)
>>> img3.hexdump()
```

6.4 Join vs. no-join

Adjacent sections are merged by default (`join=True`). Pass `join=False` to keep boundaries intact:

```
>>> s0 = Section(0x100, range(1, 9))
>>> s1 = Section(0x108, range(9, 17))

>>> img_joined = Image([s0, s1])            # join=True (default)
>>> img_nojoin = Image([s0, s1], join=False)

>>> len(img_joined)
1
>>> len(img_nojoin)
2

>>> img_joined.hexdump()
00000100  01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10  |................|
---------------
       16 bytes
---------------
```

6.5 Typed access — scalars

```
>>> img = Image([Section(0x1000, bytes(64))])

>>> img.write_numeric(0x1000, 0x10203040, "uint32_be")
>>> img.write_numeric(0x1004, 0x50607080, "uint32_le")
>>> img.read_numeric(0x1000, "uint32_be")
270544960
>>> img.read_numeric(0x1004, "uint32_le")
1348861056
```

6.6 Typed access — arrays

```
>>> img.write_numeric_array(0x1010, [0x1000, 0x2000, 0x3000], "uint16_le")
>>> img.read_numeric_array(0x1010, 3, "uint16_le")
[4096, 8192, 12288]
```

6.7 Typed access — strings

```
>>> img.write_string(0x1020, "objutils")
>>> img.read_string(0x1020)
'objutils'
```

6.8 ASAM scalars

ASAM helper methods accept ASAM/A2L type names (`ULONG`, `UWORD`, ...) and ECU byte-order terms (`MSB_FIRST`, `MSB_LAST`, `MSB_FIRST_MSW_LAST`, ...):

```
>>> img = Image([Section(0x3000, bytes(64))])

>>> img.write_asam_numeric(0x3000, 0x11223344, "ULONG", "MSB_FIRST")
>>> img.write_asam_numeric(0x3004, 0x11223344, "ULONG", "MSB_FIRST_MSW_LAST")
>>> img.write_asam_numeric(0x3008, 0x11223344, "ULONG", "MSB_LAST_MSW_FIRST")

>>> img.read_asam_numeric(0x3000, "ULONG", "MSB_FIRST")
287454020
>>> img.read_asam_numeric(0x3004, "ULONG", "MSB_FIRST_MSW_LAST")
287454020
>>> img.read_asam_numeric(0x3008, "ULONG", "MSB_LAST_MSW_FIRST")
287454020
```

6.9 ASAM strings

```
>>> img.write_asam_string(0x3010, "MOTOR",    "ASCII")
>>> img.write_asam_string(0x3020, "Drehzahl", "UTF8")

>>> img.read_asam_string(0x3010, "ASCII")
'MOTOR'
>>> img.read_asam_string(0x3020, "UTF8")
'Drehzahl'
```

6.10 ASAM numeric arrays

`length` is the **element count**:

```
>>> img = Image([Section(0x6000, bytes(32))])
>>> img.write_asam_numeric_array(0x6000, [0x11223344, 0x55667788], "ULONG", "MSB_LAST_
↪MSW_FIRST")
>>> img.read_asam_numeric_array(0x6000, 2, "ULONG", "MSB_LAST_MSW_FIRST")
(287454020, 1432778632)
```

Verify the raw bytes to understand word-swap layout:

```
>>> img.read(0x6000, 8)
b'\x33\x44\x11\x22\x77\x88\x55\x66'
```

6.11 ASAM ndarrays (NumPy)

`length` is the **element count** for `read_asam_ndarray`. `shape` uses **ASAM** dimension order `(X, Y, Z, ...)`.

```
>>> import numpy as np
>>> img = Image([Section(0x5000, bytes(64))])

>>> matrix = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.uint16)
>>> img.write_asam_ndarray(0x5000, matrix, "UWORD", "MSB_FIRST", index_mode="COLUMN_DIR")

>>> matrix_rt = img.read_asam_ndarray(
...     0x5000, 6, "UWORD", shape=(3, 2), index_mode="COLUMN_DIR", byte_order="MSB_FIRST"
... )
>>> np.array_equal(matrix_rt, matrix)
True
```

6.12 What next?

- See the *Tutorial* for a step-by-step guide through every feature.
- See *HOW-TOs* for task-oriented recipes.
- See *Scripts provided by objutils* for all CLI tools and their options.
- See *objutils* for the full auto-generated API reference.

CHAPTER

SEVEN

OBJUTILS

7.1 objutils package

7.1.1 Subpackages

objutils.elf package

Subpackages

Submodules

objutils.elf.attributes module

objutils.elf.defs module

objutils.elf.elfcons module

objutils.elf.relocs module

objutils.elf.testblocks module

Module contents

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search