tamp

Release 0.0.0

Brian Pugh

Jun 15, 2026

CONTENTS:

1 Features 3

2 Installation 5
- 2.1 Desktop Python . . . 5
- 2.2 MicroPython . . . 5
- 2.3 C . . . 6

3 Usage 7
- 3.1 CLI . . . 7
- 3.2 Python . . . 8

4 Benchmark 9
- 4.1 Compression Ratio . . . 9
- 4.2 Memory Usage . . . 10
- 4.3 Runtime . . . 11
- 4.4 Binary Size . . . 11
- 4.5 Acknowledgement . . . 12

5 Installation 13
- 5.1 Desktop Python . . . 13
- 5.2 MicroPython . . . 13
- 5.3 C . . . 14

6 Specification 15
- 6.1 Stream Header . . . 15
- 6.2 Header Byte 2 . . . 15
- 6.3 Stream Encoding/Decoding . . . 16

7 Custom Dictionary 23
- 7.1 When to Use Custom Dictionaries . . . 23
- 7.2 Improving JSON Compression . . . 24
- 7.3 Performance Considerations . . . 25

8 Python API 29

9 C Library 37
- 9.1 Compile-Time Flags . . . 37
- 9.2 Overview . . . 39
- 9.3 Compressor . . . 39
- 9.4 Decompressor . . . 45
- 9.5 Callbacks . . . 46

9.6 Stream API 47

10 JavaScript/TypeScript API **53**
10.1 Installation 53
10.2 Basic Usage 53
10.3 Error Handling 55
10.4 Custom Configuration 56
10.5 Progress Callbacks 56
10.6 Utility Functions 57
10.7 Node.js and Browser Support 58

11 Rust **59**
11.1 Documentation 59
11.2 Installation 59
11.3 Example Usage 59

12 Migrating to v2 **61**
12.1 Format Changes 61
12.2 C Library 62
12.3 Installation 63
12.4 CLI 63
12.5 Python 64
12.6 JavaScript/WASM 64

Index **65**

Documentation: https://tamp.readthedocs.io/en/latest/

Source Code: https://github.com/BrianPugh/tamp

Online Demo: https://brianpugh.github.io/tamp

Tamp is a low-memory, DEFLATE-inspired lossless compression library optimized for embedded and resource-constrained environments.

Tamp delivers the highest data compression ratios, while using the least amount of RAM and firmware storage.

CHAPTER ONE

FEATURES

- Various language implementations available:
  - Pure Python reference:
    - `tamp/__init__.py`, `tamp/compressor.py`, `tamp/decompressor.py`
    - `pip install tamp` will use a python-bound C implementation optimized for speed.
  - Micropython:
    - Native Module.
      - `mpy_bindings/`
  - C library:
    - `tamp/_c_src/`
  - Javascript/Typescript via Emscripten WASM.
    - `wasm/`
  - Unofficial rust bindings.
    - See documentation here.
- High compression ratios, low memory use, and fast.
- Compact compression and decompression implementations.
  - Compiled C library is <5KB (compressor + decompressor).
- Mid-stream flushing.
  - Allows for submission of messages while continuing to compress subsequent data.
- Customizable dictionary for greater compression of small messages.
- Fuzz tested with libFuzzer + AddressSanitizer/UBSan.
- Convenient CLI interface.

CHAPTER

TWO

INSTALLATION

Tamp contains several implementations:

1. A reference desktop CPython implementation that is optimized for readability (and **not** speed).
2. A Micropython Native Module implementation (fast).
3. A C implementation (with python bindings) for accelerated desktop use and to be used in C projects (very fast).
4. A JavaScript/TypeScript implementation via Emscripten WASM (see `wasm/`).

This section instructs how to install each implementation.

2.1 Desktop Python

The Tamp library requires Python `>=3.9` and can be installed via:

```
pip install tamp
```

To also install the `tamp` command line tool:

```
pip install tamp[cli]
```

2.2 MicroPython

2.2.1 MicroPython Native Module

Tamp provides pre-compiled [native modules]{.title-ref} that are easy to install, are small, and are incredibly fast.

Download the appropriate `.mpy` file from the release page.

- Match the micropython version.
- Match the architecture to the microcontroller (e.g. `armv6m` for a pi pico).

Rename the file to `tamp.mpy` and transfer it to your board. If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = "https://github.com/BrianPugh/tamp/releases/download/v1.7.0/tamp-1.7.0-mpy1.23-
↪armv6m.mpy"
```

2.3 C

Copy the `tamp/_c_src/tamp` folder into your project. For more information, see the documentation.

CHAPTER

THREE

USAGE

Tamp works on desktop python and micropython. On desktop, Tamp can be bundled with the `tamp` command line tool for compressing and decompressing tamp files. Install with `pip install tamp[cli]`.

3.1 CLI

3.1.1 Compression

Use `tamp compress` to compress a file or stream. If no input file is specified, data from stdin will be read. If no output is specified, the compressed output stream will be written to stdout.

```
$ tamp compress --help
Usage: tamp compress [ARGS] [OPTIONS]

Compress an input file or stream.

─ Parameters␣
↪─────────────────────────────────────────────────────────────────────────
│ INPUT,--input    -i  Input file to compress. Defaults to stdin.             ␣
↪     │
│ OUTPUT,--output  -o  Output compressed file. Defaults to stdout.            ␣
↪     │
│ --window         -w  Number of bits used to represent the dictionary window. [default:␣
↪10] │
│ --literal        -l  Number of bits used to represent a literal. [default: 8]        ␣
↪     │
─────────────────────────────────────────────────────────────────────────────────────
```

Example usage:

```
tamp compress enwik8 -o enwik8.tamp  # Compress a file
echo "hello world" | tamp compress | wc -c  # Compress a stream and print the compressed␣
↪size.
```

The following options can impact compression ratios and memory usage:

- `window` - `2^window` plaintext bytes to look back to try and find a pattern. A larger window size will increase the chance of finding a longer pattern match, but will use more memory, increase compression time, and cause each pattern-token to take up more space. Try smaller window values if compressing highly repetitive data, or short messages.

- `literal` - Number of bits used in each plaintext byte. For example, if all input data is 7-bit ASCII, then setting this to 7 will improve literal compression ratios by 11.1%. The default, 8-bits, can encode any binary data.

3.1.2 Decompression

Use `tamp decompress` to decompress a file or stream. If no input file is specified, data from stdin will be read. If no output is specified, the compressed output stream will be written to stdout.

```
$ tamp decompress --help
Usage: tamp decompress [ARGS] [OPTIONS]

Decompress an input file or stream.

─ Parameters␣
↪──────────────────────────────────────────────────────────────────────────────
│ INPUT,--input    -i  Input file to decompress. Defaults to stdin.            ␣
↪     │
│ OUTPUT,--output  -o  Output decompressed file. Defaults to stdout.           ␣
↪     │
────────────────────────────────────────────────────────────────────────────────────────────
```

Example usage:

```
tamp decompress enwik8.tamp -o enwik8
echo "hello world" | tamp compress | tamp decompress
```

3.2 Python

The python library can perform one-shot compression, as well as operate on files/streams.

```python
import tamp

# One-shot compression
string = b"I scream, you scream, we all scream for ice cream."
compressed_data = tamp.compress(string)
reconstructed = tamp.decompress(compressed_data)
assert reconstructed == string

# Streaming compression
with tamp.open("output.tamp", "wb") as f:
    for _ in range(10):
        f.write(string)

# Streaming decompression
with tamp.open("output.tamp", "rb") as f:
    reconstructed = f.read()
```

CHAPTER

FOUR

BENCHMARK

In the following section, we compare Tamp against:

- zlib, a python builtin gzip-compatible DEFLATE compression library.
- heatshrink, a data compression library for embedded/real-time systems. Heatshrink has similar goals as Tamp.

All of these are LZ-based compression algorithms, and tests were performed using a 1KB (10 bit) window. Since zlib already uses significantly more memory by default, the lowest memory level (`memLevel=1`) was used in these benchmarks. It should be noted that higher zlib memory levels will having greater compression ratios than Tamp. Currently, there is no micropython-compatible zlib or heatshrink compression implementation, so these numbers are provided simply as a reference.

4.1 Compression Ratio

The following table shows compression algorithm performance over a variety of input data sourced from the Silesia Corpus and Enwik8. This should give a general idea of how these algorithms perform over a variety of input data types.

dataset	raw	tamp	tamp (LazyMatching)	zlib	heatshrink
enwik8	100,000,000	51,016,917	**50,625,930**	56,205,166	56,110,394
RPI_PICO (.uf2)	667,648	**289,454**	290,577	303,763	-
silesia/dickens	10,192,446	5,538,353	**5,502,834**	6,049,169	6,155,768
silesia/mozilla	51,220,480	24,413,362	**24,229,925**	25,104,966	25,435,908
silesia/mr	9,970,564	4,520,091	**4,391,864**	4,864,734	5,442,180
silesia/nci	33,553,445	6,824,403	6,772,307	**5,765,521**	8,247,487
silesia/ooffice	6,152,192	3,773,003	**3,755,046**	4,077,277	3,994,589
silesia/osdb	10,085,684	8,466,875	**8,464,328**	8,625,159	8,747,527
silesia/reymont	6,627,202	2,818,554	**2,788,774**	2,897,661	2,910,251
silesia/samba	21,606,400	8,383,534	**8,346,076**	8,862,423	9,223,827
silesia/sao	7,251,944	6,136,077	**6,100,061**	6,506,417	6,400,926
silesia/webster	41,458,703	18,146,641	**18,010,981**	20,212,235	19,942,817
silesia/x-ray	8,474,240	7,509,449	7,404,794	**7,351,750**	8,059,723
silesia/xml	5,345,280	1,472,562	**1,455,641**	1,586,985	1,665,179

Tamp outperforms both heatshrink and zlib on most datasets, winning 12 out of 14 benchmarks. This is while using around 10x less memory than zlib during both compression and decompression (see next section).

Lazy Matching is a simple technique to improve compression ratios at the expense of CPU while requiring very little code. One can expect **50-75%** more CPU usage for modest compression gains (around 0.5 - 2.0%). Because of this trade-off, it is disabled by default; however, in applications where we want to compress once on a powerful machine

(like a desktop/server) and decompress on an embedded device, it may be worth it to spend a bit more compute. Lazy matched compressed data is the exact same format; it appears no different to the tamp decoder.

4.1.1 Ablation Study

The following table shows the effect of the `extended` and `lazy_matching` compression parameters across all benchmark datasets (`window=10`, `literal=8`).

dataset	raw	Baseline	+lazy	+extended	+lazy +extended
enwik8	100,000,000	51,635,633	51,252,694 (0.7%)	51,016,917 (1.2%)	50,625,930 (2.0%)
RPI_PICO (.uf2)	667,648	331,310	329,893 (0.4%)	289,454 (12.6%)	290,577 (12.3%)
silesia/dickens	10,192,446	5,546,761	5,511,681 (0.6%)	5,538,353 (0.2%)	5,502,834 (0.8%)
silesia/mozilla	51,220,480	25,121,385	24,937,036 (0.7%)	24,413,362 (2.8%)	24,229,925 (3.5%)
silesia/mr	9,970,564	5,027,032	4,888,930 (2.7%)	4,520,091 (10.1%)	4,391,864 (12.6%)
silesia/nci	33,553,445	8,643,610	8,645,399 (+0.0%)	6,824,403 (21.0%)	6,772,307 (21.6%)
silesia/ooffice	6,152,192	3,814,938	3,798,393 (0.4%)	3,773,003 (1.1%)	3,755,046 (1.6%)
silesia/osdb	10,085,684	8,520,835	8,518,502 (0.0%)	8,466,875 (0.6%)	8,464,328 (0.7%)
silesia/reymont	6,627,202	2,847,981	2,820,948 (0.9%)	2,818,554 (1.0%)	2,788,774 (2.1%)
silesia/samba	21,606,400	9,102,594	9,061,143 (0.5%)	8,383,534 (7.9%)	8,346,076 (8.3%)
silesia/sao	7,251,944	6,137,755	6,101,747 (0.6%)	6,136,077 (0.0%)	6,100,061 (0.6%)
silesia/webster	41,458,703	18,694,172	18,567,618 (0.7%)	18,146,641 (2.9%)	18,010,981 (3.7%)
silesia/x-ray	8,474,240	7,510,606	7,406,001 (1.4%)	7,509,449 (0.0%)	7,404,794 (1.4%)
silesia/xml	5,345,280	1,681,687	1,672,827 (0.5%)	1,472,562 (12.4%)	1,455,641 (13.4%)

The `extended` parameter enables additional Huffman codes for longer pattern matches, which significantly improves compression on datasets with many long repeating patterns (e.g., nci, samba, xml). Extended support was added in v2.0.0.

4.2 Memory Usage

The following table shows approximately how much memory each algorithm uses during compression and decompression.

	Compression	Decompression
Tamp	(1 << windowBits)	(1 << windowBits)
ZLib	(1 << (windowBits + 2)) + 7KB	(1 << windowBits) + 7KB
Heatshrink	(1 << (windowBits + 1))	(1 << (windowBits + 1))
Deflate (micropython)	(1 << windowBits)	(1 << windowBits)

All libraries have a few dozen bytes of overhead in addition to the primary window buffer, but are implementation-specific and ignored for clarity here. Tamp uses significantly less memory than ZLib, and half the memory of Heatshrink.

4.3 Runtime

As a rough benchmark, here is the performance (in seconds) of these different compression algorithms on the 100MB enwik8 dataset. These tests were performed on an M3 Macbook Air.

	Compression (s)	Decompression (s)
Tamp (Pure Python Reference)	136.2	105.0
Tamp (C bindings)	5.45	0.544
ZLib	3.65	0.578
Heatshrink (with index)	4.42	0.67
Heatshrink (without index)	27.40	0.67

Heatshrink v0.4.1 was used in these benchmarks. When heathshrink uses an index, an additional `(1 << (windowBits + 1))` bytes of memory are used, resulting in 4x more memory-usage than Tamp. Tamp could use a similar indexing to increase compression speed, but has chosen not to to focus on the primary goal of a low-memory compressor.

To give an idea of Tamp's speed on an embedded device, the following table shows compression/decompression in **bytes/second of the first 100KB of enwik8 on a pi pico (rp2040)** at the default 125MHz clock rate. The C benchmark **does not** use a filesystem nor dynamic memory allocation, so it represents the maximum speed Tamp can achieve. In all tests, a 1KB window (10 bit) was used.

	Compression (bytes/s)	Decompression (bytes/s)
Tamp (Micropython Native Module)	31,328	990,099
Tamp (C)	36,127	1,400,600
Deflate (micropython builtin)	6,885	294,985

Tamp resulted in a **50841** byte archive, while Micropython's builtin `deflate` resulted in a larger, **59442** byte archive.

4.4 Binary Size

To give an idea on the resulting binary sizes, Tamp and other libraries were compiled for the Pi Pico (`armv6m`). All libraries were compiled with `-O3`. Numbers reported in bytes. Tamp sizes were measured using `arm-none-eabi-gcc` 15.2.1 and MicroPython v1.27, and can be regenerated with `make binary-size`.

	Compressor	Decompressor	Compressor + Decompressor
Tamp (MicroPython Native)	4700	4347	8024
Tamp (C, no extended, no stream)	1754	1656	3172
Tamp (C, no extended)	2036	1894	3692
Tamp (C, extended, no stream)	2838	2452	5052
Tamp (C, extended)	3120	2690	5572
Heatshrink (C)	2956	3876	6832
uzlib (C)	2355	3963	6318

Tamp C "extended" includes `tamp_compressor_compress_and_flush`. Tamp C includes a high-level stream API by default. Even with `no stream`, Tamp includes buffer-looping functions (like `tamp_compressor_compress`) that

Heatshrink lacks (Heatshrink only provides poll/sink primitives).

4.5 Acknowledgement

- Thanks @BitsForPeople for the esp32-optimized compressor implementation.

CHAPTER

FIVE

INSTALLATION

Tamp contains 4 implementations:

1. A reference desktop CPython implementation that is optimized for readability (and **not** speed).
2. A Micropython Native Module implementation (fast).
3. A Micropython Viper implementation (not recommended, please use Native Module).
4. A C implementation (with python bindings) for accelerated desktop use and to be used in C projects (very fast).

5.1 Desktop Python

The Tamp library and CLI requires Python `>=3.9` and can be installed via:

```
pip install tamp
```

To install directly from github, you can run:

```
python -m pip install git+https://github.com/BrianPugh/tamp.git
```

For development, its recommended to use Poetry:

```
git clone https://github.com/BrianPugh/tamp.git
cd tamp
poetry install
```

5.2 MicroPython

5.2.1 Native Module

Tamp provides pre-compiled *native modules* that are easy to install, are small, and are incredibly fast.

Download the appropriate `.mpy` file from the release page.

- Match the micropython version.
- Match the architecture to the microcontroller (e.g. `armv6m` for a pi pico).

Rename the file to `tamp.mpy` and transfer it to your board. If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = "https://github.com/BrianPugh/tamp/releases/download/v1.7.0/tamp-1.7.0-mpy1.23-
↪armv6m.mpy"
```

5.2.2 Viper

NOT RECOMMENDED, PLEASE USE NATIVE MODULE

For micropython use, there are 3 main files:

1. `tamp/__init__.py` - Always required.
2. `tamp/decompressor_viper.py` - Required for on-device decompression.
3. `tamp/compressor_viper.py` - Required for on-device compression.

For example, if on-device decompression isn't used, then do not include `decompressor_viper.py`. If manually installing, just copy these files to your microcontroller's `/lib/tamp` folder.

If using mip, tamp can be installed by specifying the appropriate `package-*.json` file.

```
mip install github:brianpugh/tamp  # Defaults to package.json: Compressor & Decompressor
mip install github:brianpugh/tamp/package-compressor.json  # Compressor only
mip install github:brianpugh/tamp/package-decompressor.json  # Decompressor only
```

If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = [
   "https://github.com/BrianPugh/tamp/blob/main/tamp/__init__.py",
   "https://github.com/BrianPugh/tamp/blob/main/tamp/compressor_viper.py",
   "https://github.com/BrianPugh/tamp/blob/main/tamp/decompressor_viper.py",
]
```

5.3 C

Copy the `tamp/_c_src/tamp` folder into your project. For more information, see *C Library*.

CHAPTER

SIX

SPECIFICATION

Tamp uses a single-pass DEFLATE-like algorithm that is optimized for a fast, simple implementation in micropython. Trade-offs were made between code-complexity, speed, memory-usage, and compression-ratio. This document describes the compression algorithm, design choices, and the data format. Tamp outputs a continuous bit stream, where the most-significant-bit (MSb) comes first.

6.1 Stream Header

Tamp has a single header byte described in the following table. The locations are zero index from the beginning of the stream. The bit-location 0 is equivalent to typical MSb position 7 of the first byte.

Bits	Name	Description
[7,6,5	window	Number of bits, minus 8, used to represent the size of the shifting window. e.g. A 12-bit window is encoded as the number 4, `0b100`. This means the smallest window is 256 bytes, and largest is 32768.
[4,3]	literal_size	Number of bits, minus 5, in a literal. For example, `0b11` represents a standard 8 bit (1 byte) literal.
[2]	custom_dictic	A custom dictionary initialization method was used and must be provided at decompression.
[1]	extended	Enables extended format features (RLE, extended match encoding). Generally improves compression, introduced in tamp v2.0.0.
[0]	more_heac	Next byte is header byte 2 (see *Header Byte 2*). Implies `dictionary_reset`: stream may contain double-FLUSH resets. Old decompressors reject the stream at the header.

6.2 Header Byte 2

Present only when `more_header` (bit 0 of byte 1) is set. All 8 bits are reserved for future use and must be `0`. Decompressors must reject non-zero values with `TAMP_INVALID_CONF`.

6.3 Stream Encoding/Decoding

After the header bytes is the data stream. The datastream is written in bits, so all data is packed tightly with no padding. At a high level, Tamp applies LZSS compression, followed by a fixed, pre-defined Huffman coding for representing the match length.

6.3.1 LZSS

Tamp uses a *slightly* modified LZSS compression algorithm. Modifications are made to make the implementation simpler/faster.

1. Initialize a ring_buffer of size `1 << window` defined in the header. See Dictionary Initialization for initialization details.
2. Starting at the beginning of the plaintext, find the longest match existing in the dictionary. If no pattern (<2 character match) is found, output a literal. If a pattern is detected
   a. literal: `0b1 | literal`. The first bit (`1`) represents that the following bits represent a literal. The `literal` is `literal_size` bits long.
   b. token: `0b0 | length-huffman-code | offset`. The first bit (`0`) represents that the following bits represent a token. The length of the pattern match is encoded via a pre-defined static Huffman code. Finally, the `offset` is `window` bits long, and points at the offset from the beginning of the dictionary buffer to the pattern. The shortest pattern-length is either going to be 2 or 3 bytes, depending on `window` and `literal` parameters. The shortest pattern-length encoding must be shorter than an equivalent stream of literals. In the basic (non-extended) format, the longest pattern-length is the minimum pattern-length plus 13. When the `extended` flag is set, longer matches are possible via extended match encoding.

Classically, the `offset` is from the current position in the buffer. Doing so results in the `offset` distribution slightly favoring smaller numbers. Intuitively, it makes sense that patterns are more likely to occur closer to the current text. A proposed idea was to include the `offset` most-significant bit with the length-huffman-code. Unfortuneately, the probability distribution wasn't biased enough to increase compression. For this reason, we just encode the simple offset from the beginning of the dictionary. This is easier to implement and has the potential to execute quicker.

Similarly, attempts to include the `is_literal` flag in the huffman coding did not increase compression. An explicit `is_literal` flag made the code faster and simpler.

6.3.2 Dictionary Initialization

For short messages, having a better initial dictionary can help improve compression ratios. The amount of improvement would be dependent on the type of data being compressed. Given that the contents of raw-binary data could be anything, we chose to focus on improving typical english text. In order to save device space, a whole dictionary is not saved to disk. Instead, we select 16 common characters appropriate for the `literal` bit width and pseudo-randomly fill up the dictionary. We use the XorShift32 pseudo-random number generator due to it's good randomness characteristics, and simple implementation.

The character table is chosen based on the `literal` parameter:

- **literal=7 or 8**: 16 common english text and markup characters:

```
 \x000ei>to<ans\nr/.
^there is a "space" there.

(or as explicit hex values)
0x20 0x00 0x30 0x65 0x69 0x3E 0x74 0x6F 0x3C 0x61 0x6E 0x73 0x0A 0x72 0x2F 0x2E
```

- **literal=5 or 6**: The 16 most common english characters (`" etaoinshrdlcumw"`) downshifted to the target bit width by masking with `(1 << literal) - 1`. Since ASCII encodes letter position in the low bits, this preserves alphabetic identity.

> **Warning**
>
> When the `extended` header flag is *not* set (v1 format), `literal` is always treated as 8 for backwards compatibility, regardless of the configured value.

The chosen seed, 3758097560, was discovered experimentally to give typically good results for 7/8 bit data.

All of this amounts to a few percent compression improvement for short messages.

```
def _xorshift32(seed):
    while True:
        seed ^= (seed << 13) & 0xFFFFFFFF
        seed ^= (seed >> 17) & 0xFFFFFFFF
        seed ^= (seed << 5) & 0xFFFFFFFF
        yield seed


def initialize_dictionary(size, literal=8):
    if literal <= 5:
        chars = bytes([c & 0x1F for c in b" etaoinshrdlcumw"])
    elif literal <= 6:
        chars = bytes([c & 0x3F for c in b" etaoinshrdlcumw"])
    else:
        chars = b" \x000ei>to<ans\nr/."

    def _gen_stream(xorshift32):
        for _ in range(size >> 3):
            value = next(xorshift32)
            yield chars[value & 0x0F]
            yield chars[value >> 4 & 0x0F]
            yield chars[value >> 8 & 0x0F]
            yield chars[value >> 12 & 0x0F]
            yield chars[value >> 16 & 0x0F]
            yield chars[value >> 20 & 0x0F]
            yield chars[value >> 24 & 0x0F]
            yield chars[value >> 28 & 0x0F]

    return bytearray(_gen_stream(_xorshift32(3758097560)))
```

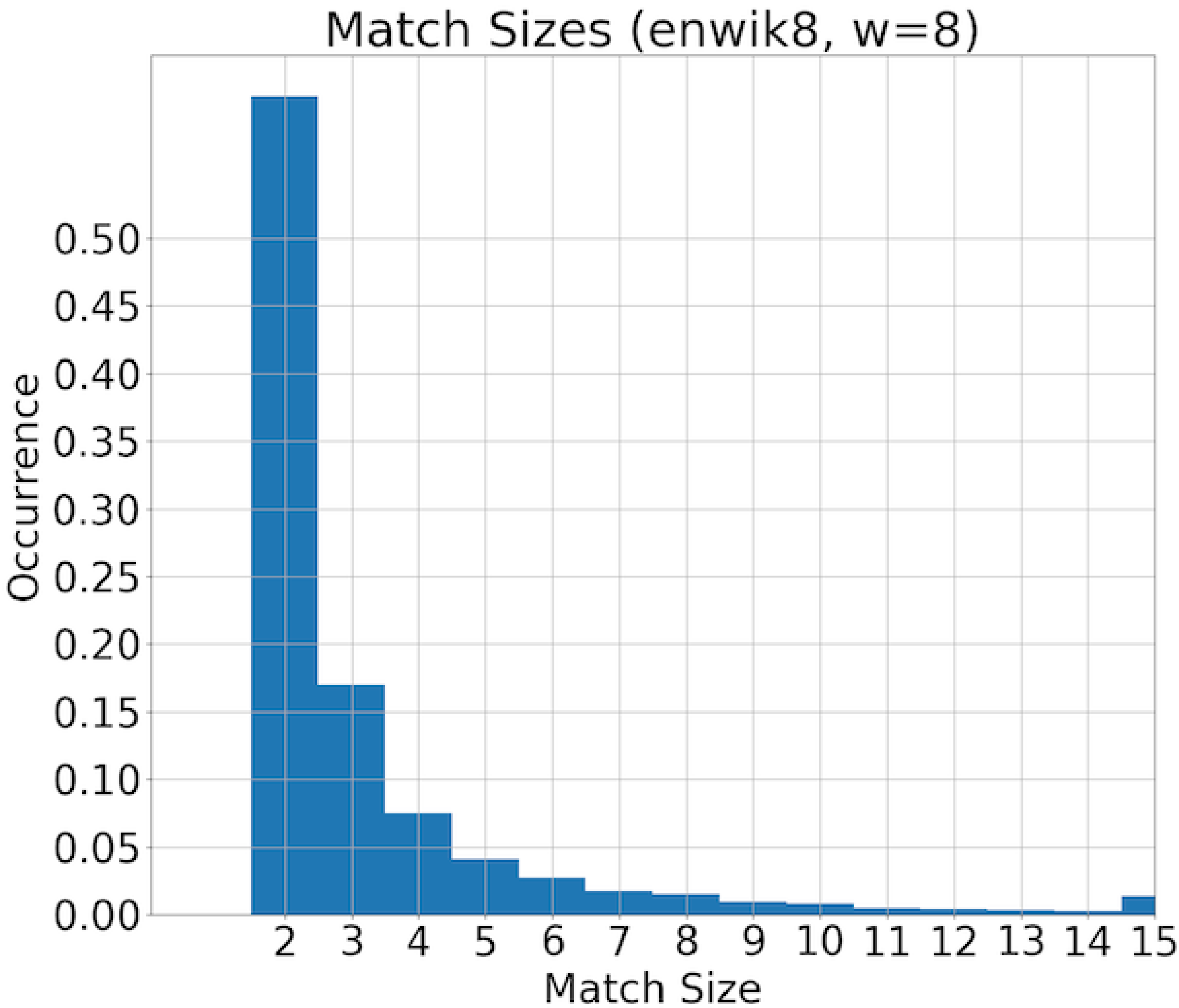
6.3.3 Huffman Coding

Huffman coding encodes high-probability values with less bits, and less-likely values with more bits. In order for huffman coding to work, no encoding is allowed to be a prefix of another encoding. If all values have equal probability, simple binary encoding is more efficient.

The following maps the pattern-size (to be added to the minimum pattern-length) to the bits representing the huffman code.

```
huffman_coding = {
    0: 0b0,
    1: 0b11,
    2: 0b1000,
    3: 0b1011,
    4: 0b10100,
    5: 0b100100,
    6: 0b100110,
    7: 0b101011,
    8: 0b1001011,
    9: 0b1010100,
    10: 0b10010100,
    11: 0b10010101,
    12: 0b10101010,
    13: 0b100111,
    "FLUSH": 0b10101011,
}
```

The match-size probabilities that generated this table were generated over the enwik8 dataset. This huffman coding was chosen such that the longest huffman code is 8 bits long, making it easier to store and index into. The maximum match-size is more likely than the second-highest match-size because all match-sizes greater than the maximum size get down-mapped.



For any given huffman coding schema, a equivalent coding can be obtained by inverting all the bits (reflecting the huffman tree). The single-bit, most common code `0b0` representing a pattern-size 2 is intentionally represented as `0b0` instead of `0b1`. This makes the MSb of all other codes be 1, simplifying the decoding procedure because the number of bits read doesn't strictly have to be recorded.

6.3.4 Extended Format (v2.0.0+)

When the `extended` header bit is set, two additional token types are available: RLE (Run-Length Encoding) and Extended Match. These use Huffman symbols 12 and 13 respectively, which in the basic format would represent match sizes `min_pattern_size + 12` and `min_pattern_size + 13`.

Extended Huffman Encoding

Both RLE and Extended Match use a secondary Huffman encoding to represent their payload values. This encoding combines a Huffman code (without the literal flag) with trailing bits:

1. Read the Huffman symbol (12 for RLE, 13 for Extended Match) with the literal flag (`0b0`).
2. Decode an additional Huffman code (reusing the same table, but without the leading literal flag bit). In this secondary context, symbol 14 (the FLUSH bit pattern `0b10101011`) is a valid value, giving Huffman indices 0 through 14.
3. Read trailing bits (4 bits for RLE, 3 bits for Extended Match).
4. Combine: `value = (huffman_index << trailing_bits) + trailing_bits_value`

RLE Token (Symbol 12)

RLE encodes runs of repeated bytes efficiently. The repeated byte is implicitly the last byte written to the window buffer. If no bytes have been written yet (i.e., `window_pos == 0`), the byte at position `window_size - 1` of the initial dictionary is used.

- `huffman_code[12]` = `0xAA` (9 bits including literal flag)
- `count` ranges from 2 to 241: `(14 << 4) + 15 + 2 = 241`

Window update: Only the first 8 bytes are written to the dictionary (no wrap-around). If fewer than 8 bytes remain before the end of the window buffer, only those bytes are written. Writing only 8 bytes preserves useful dictionary content for future pattern matches, since filling the window with a single repeated byte would reduce match opportunities.

```
RLE Token Structure:
+---+------------+---------------------------+
| 0 | huffman[12]| extended_huffman(count - 2) |
+---+------------+---------------------------+
|1b |   8 bits   | (1 ~ 8) + 4 bits           |
+---+------------+---------------------------+
```

Extended Match Token (Symbol 13)

Extended Match allows pattern matches longer than the basic format's maximum of `min_pattern_size + 13`. It is used when a match exceeds `min_pattern_size + 11`.

- `huffman_code[13]` = `0x27` (7 bits including literal flag)
- `offset` is `window` bits, pointing to the start of the pattern
- Maximum extra size: `(14 << 3) + 7 + 1 = 120`
- Maximum total match size: `min_pattern_size + 11 + 120 = min_pattern_size + 131`

The `-12` offset ensures extended matches start at `min_pattern_size + 12`, leaving symbols 0-11 for basic matches (0-11 maps to `min_pattern_size` through `min_pattern_size + 11`).

Window constraints: The source pattern cannot span past the window buffer boundary; the compressor terminates extended matches early if they would cross this boundary. Similarly, destination writes do not wrap-around; only bytes up to the end of the window buffer are written. This simplifies implementation while having minimal impact on compression ratio (approximately 0.02% loss).

```
Extended Match Token Structure:
+---+------------+------------------------+--------+
| 0 | huffman[13]| extended_huffman(sz)     | offset |
+---+------------+------------------------+--------+
|1b |   6 bits   | (1 ~ 8) + 3 bits         | window |
+---+------------+------------------------+--------+

Where sz = match_size - min_pattern_size - 12
```

Flush Symbol

A special FLUSH symbol is encoded as the least likely Huffman code. In many compression algorithms, a `flush()` can only be called at the end of the compression stream, and the compressor cannot be used anymore. In microcontroller applications, the user may want to flush the compressor buffer while still continuing to compress more data. Examples include:

1. Flushing a chunk of logs to disk to prepare if power is removed.
2. Pushing a chunk of collected data to a remote server.

Since the output bit stream must be byte-aligned for writing, there may be up to 7 pending bits that have not yet formed a complete byte. Invoking the `flush` method can have one of two results:

1. If the output is already byte-aligned and `more_header` is not set, no action is performed.
2. If there are pending bits, or if `more_header` is set, the FLUSH Huffman code is written. No `offset` bits are written following the FLUSH code. The remaining bits are zero-padded to the next byte boundary.

On reading, if a FLUSH is read, the reader discards the remaining bits up to the next byte boundary. When `more_header` is set, a FLUSH is **always** emitted (even when byte-aligned) to support append mode (see *Dictionary Reset (Double-FLUSH)*). In the worst case (1 pending bit), a FLUSH symbol (9 bits) and 6 padding bits are written, adding 15 bits of overhead to the output stream.

Dictionary Reset (Double-FLUSH)

Dictionary reset allows appending new compressed data to an existing stream without retaining the previous compressor state. After a reset, both sides start fresh with a default dictionary.

When the `more_header` (header byte 1, bit 0) is set, two consecutive FLUSH tokens signal a dictionary reset. Without `more_header`, double-FLUSHes are treated as two ordinary flushes. Old decompressors (<2.1.0) reject `more_header` streams at the header, preventing silent corruption.

On the compressor side, `reset_dictionary` flushes pending data, writes exactly two consecutive FLUSHes, then re-initializes the window and all internal state.

On the decompressor side: 1. Re-initializes the window (see *Dictionary Initialization*). Custom dictionaries cannot be supplied at this time. 2. Resets window position to zero. 3. Continues decompressing with a fresh dictionary.

6.3.5 Miscellaneous

No terminating character is builtin. Tamp relies on external framing (such as from the filesystem) to know when the data stream is complete. The final byte of a stream is zero-padded. The maximum padding is 7 zero bits.

CHAPTER

SEVEN

CUSTOM DICTIONARY

Tamp is in the LZ family of compression algorithms. Like other LZ-based compressors, Tamp builds a sliding window (AKA dictionary) of recently seen data to find repeating patterns. For short messages, this window is relatively uninitialized, leading to relatively poor compression for the first few hundred bytes of a stream.

Tamp attempts to mitigate this by initializing this sliding window with a deterministic random combinations of letters (see *Dictionary Initialization*). The character table used for initialization depends on the `literal` bit width: for `literal=7` or `8`, common english text and markup characters are used; for `literal=5` or `6`, common english letters downshifted to the target bit width are used instead.

For the default `literal=8`, the first bytes of the dictionary look like this (where `\x00` is NULL; `\n` is newline):

```
\x00.//r.0. t>\n/>snas.trnr i\x00r/a\x00snat./.r\x00i o.s tneo>.as>\na.ta\x00 aa\x00\x00\
↪x000oe ri\x00a>eatsi\n.\ni.str\n//snesr.ost
```

This "random" series of bytes contains many short patterns that are common in english text.

You, the developer, may have better insight on what the first few hundred bytes of your message may look like. In these situations, having a shared custom dictionary can improve compression efficiency. Custom dictionaries solve this problem by pre-populating the compression window with commonly occurring data patterns, allowing the algorithm to find matches immediately rather than having to "learn" patterns from recent input.

Note

Custom dictionaries must be externally supplied to **both** the compressor and decompressor. The dictionary is not embedded in the compressed data, so both parties **must** have access to the same custom dictionary for successful compression and decompression.

7.1 When to Use Custom Dictionaries

Custom dictionaries are most beneficial for:

- **Short predictable messages** - Where the sliding window doesn't have enough data to find patterns
- **Structured data** - JSON, XML, or other formats with repeating field names and structures
- **Domain-specific data** - Messages with known common prefixes, suffixes, or patterns
- **Protocol messages** - Network protocols or APIs with fixed headers and field names

If your use-case does not fit these situations, a custom dictionary may be more trouble than it is worth.

7.2 Improving JSON Compression

JSON is an ideal candidate for custom dictionaries because it contains many repeated elements:

- Field names that appear in every message
- Common values like `true`, `false`, `null`
- Structural characters: `{`, `}`, `[`, `]`, `:`
- Common string patterns

7.2.1 Basic Example

Consider this typical API response structure:


```
{
  "status": "success",
  "timestamp": "2024-01-15T10:30:00Z",
  "data": {
    "user_id": 12345,
    "username": "alice",
    "email": "alice@example.com",
    "is_active": true,
    "profile": {
      "first_name": "Alice",
      "last_name": "Smith",
      "created_at": "2023-06-01T09:00:00Z"
    }
  }
}
```


We can create a custom dictionary containing all the common field names and values:

```
import tamp

json_data = """\
{
    "status": "success",
    "timestamp": "2024-01-15T10:30:00Z",
    "data": {
        "user_id": 12345,
        "username": "alice",
        "email": "alice@example.com",
        "is_active": true,
        "profile": {
        "first_name": "Alice",
        "last_name": "Smith",
        "created_at": "2023-06-01T09:00:00Z"
        }
    }
}
"""
```

(continues on next page)

(continued from previous page)

```
# Initialize dictionary with Tamp's default algorithm
window_bits = 10  # 1024 bytes
dictionary = tamp.initialize_dictionary(1 << window_bits)

# Add custom JSON patterns at the end (last to get overwritten)
custom_patterns = b'{"status": "success","error": "timestamp": "data": "user_id":
↪"username": "","email": "","is_active": truefalsenull,"profile": {"first_name": "",
↪"last_name": "","created_at": ""}}[]'
dictionary[-len(custom_patterns) :] = custom_patterns

# Compress with default initialization
compressed_default = tamp.compress(json_data.encode("utf-8"))
print(
    f"Default dictionary compressed ratio: {len(json_data) / len(compressed_default):.3}"
)

# Compress with custom dictionary
compressed_custom = tamp.compress(json_data.encode("utf-8"), dictionary=dictionary)
print(
    f"Custom dictionary compressed ratio: {len(json_data) / len(compressed_custom):.3}"
)

# Decompress (must use same dictionary)
decompressed = tamp.decompress(compressed_custom, dictionary=dictionary)
```

Running this example demonstrates significantly higher compression ratios when using the custom dictionary for this short message:

```
$ python tamp-demo.py
Default dictionary compressed ratio: 1.55
Custom dictionary compressed ratio: 2.31
```

7.3 Performance Considerations

The effectiveness of a custom dictionary depends on:

- **Pattern frequency** - How often dictionary patterns appear **early** in your data
- **Dictionary size** - Larger dictionaries can hold more patterns but take up storage space
- **Message length** - Shorter messages (messages smaller than the window size) benefit most from custom dictionaries

7.3.1 Dictionary Initialization Strategy

If your custom dictionary content is shorter than the sliding window size, it's recommended to initialize the dictionary buffer using Tamp's default initialization algorithm first, then append your custom patterns at the end.

This approach provides the best of both worlds:

- The default initialization fills the window with deterministic pseudo-random data that **might** provide some additional compression.
- Your custom patterns at the end are the last to get overwritten, so they are used for longer during compression.

Python Example:

```
import tamp

# Get default initialized dictionary for window size
window_bits = 10  # 1024 bytes
dictionary = tamp.initialize_dictionary(1 << window_bits)

# Append your custom patterns at the end
custom_patterns = b'{"status":"success","error","timestamp":"data":'
dictionary[-len(custom_patterns) :] = custom_patterns

# Use the hybrid dictionary
compressed = tamp.compress(data, dictionary=dictionary)
```

JavaScript/TypeScript Example:

```
import { initializeDictionary } from '@brianpugh/tamp';

const windowBits = 10;
const dictionarySize = 1 << windowBits;  // 1024 bytes

// Initialize with Tamp's default algorithm
const dictionary = await initializeDictionary(dictionarySize);

// Add your custom patterns at the end (highest priority)
const customPatterns = new TextEncoder().encode('{"status":"success","error","timestamp":
↪"data":');
const startIndex = dictionary.length - customPatterns.length;
dictionary.set(customPatterns, startIndex);

const options = { window: windowBits, dictionary: dictionary };
const compressed = await compress(data, options);
```

7.3.2 Alternative Serialization Formats

While custom dictionaries can improve compression significantly for short messages, it is important to also mention that more efficient initial representation of your data-to-be-compressed is also important.

For example, instead of transmitting JSON data, you may want to use something much more efficient like MessagePack.

MessagePack is a binary serialization format that's more compact than JSON:

```
import json
import msgpack  # pip install msgpack
import tamp

data = {"user_id": 12345, "username": "alice", "is_active": True}

# Original JSON approach
json_data = json.dumps(data)
json_compressed = tamp.compress(json_data)

# MessagePack approach
msgpack_data = msgpack.packb(data)
msgpack_compressed = tamp.compress(msgpack_data)

print(f"JSON size: {len(json_data)} -> {len(json_compressed)} bytes")
print(f"MessagePack size: {len(msgpack_data)} -> {len(msgpack_compressed)} bytes")
```

```
$ python msgpack-demo.py
JSON size: 58 -> 52 bytes
MessagePack size: 38 -> 39 bytes  # Tamp was unable to compress msgpack; it actually␣
↪made it worse!
```

Using an appropriate, equivalent custom dictionary for both serializations gives good results:

```
import json
import msgpack
import tamp

data = {"user_id": 12345, "username": "alice", "is_active": True}

window_bits = 10  # 1024 bytes
dictionary = tamp.initialize_dictionary(1 << window_bits)

# Original JSON approach
json_data = json.dumps(data)
json_custom_dictionary = dictionary.copy()
json_patterns = b'{"user_id": 0, "username": ", "is_active": truefalse}'
json_custom_dictionary[-len(json_patterns) :] = json_patterns
json_compressed = tamp.compress(json_data, dictionary=json_custom_dictionary)

# MessagePack approach
msgpack_data = msgpack.packb(data)
msgpack_custom_dictionary = dictionary.copy()
msgpack_patterns = b"\x81\x82\x83\x84\xa7user_id\xa8username\xa9is_active\xc2\xc3"
json_custom_dictionary[-len(msgpack_patterns) :] = msgpack_patterns
```

(continues on next page)

(continued from previous page)

```
msgpack_compressed = tamp.compress(msgpack_data, dictionary=json_custom_dictionary)

print(f"JSON size: {len(json_data)} -> {len(json_compressed)} bytes")
print(f"MessagePack size: {len(msgpack_data)} -> {len(msgpack_compressed)} bytes")
```

```
$ python msgpack-demo.py
JSON size: 58 -> 22 bytes
MessagePack size: 38 -> 17 bytes
```

With the gap narrowing, it is up to the developer to make an appropriate tradeoff between system complexity, data compression, and firmware size.

CHAPTER

EIGHT

PYTHON API

class tamp.**Compressor**(*f*, *, *window: int = 10*, *literal: int = 8*, *dictionary: bytearray | None = None*, *lazy_matching: bool = False*, *extended: bool = True*, *dictionary_reset: bool = False*, *append: bool = False*)

Compresses data to a file or stream.

Parameters

- **f** (*Union[str, Path, FileLike]*) -- Path/FileHandle/Stream to write compressed data to.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is `7` for storing ascii characters where the most-significant-bit is always 0. Smaller values result in higher compression ratios for no additional computation cost. Valid range: `[5, 8]`.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with `initialize_dictionary()`
- **lazy_matching** (*bool*) -- Use roughly 50% more cpu to get 0~2% better compression.
- **append** (*bool*) -- Initialize for appending to an existing stream instead of writing a header. Writes a FLUSH token that, combined with the previous stream's trailing FLUSH, triggers a dictionary reset in the decompressor. Requires `dictionary_reset=True`.

write(*data: bytes | bytearray*) → int

Compress `data` to stream.

Parameters

data (*Union[bytes, bytearray]*) -- Data to be compressed.

Returns

Number of compressed bytes written. May be zero when data is filling up internal buffers.

Return type

int

flush(*write_token: bool = True*) → int

Flushes all internal buffers.

This compresses any data remaining in the input buffer, and flushes any remaining data in the output buffer to disk.

Parameters
write_token (*bool*) -- If appropriate, write a FLUSH token. Defaults to True.

Returns
Number of compressed bytes flushed to disk.

Return type
int

reset_dictionary() → int

Reset the dictionary and internal state, writing a double-FLUSH signal.

This allows the decompressor to re-initialize its dictionary at the corresponding point in the stream. Useful for long streams where the data characteristics change significantly.

Returns
Number of compressed bytes written.

Return type
int

Raises
ValueError -- If the compressor was not initialized with `dictionary_reset=True`.

close() → int

Flushes all internal buffers and closes the output file or stream, if tamp opened it.

Returns
Number of compressed bytes flushed to disk.

Return type
int

__enter__() → *Compressor*

Use `Compressor` as a context manager.

```
with tamp.Compressor("output.tamp") as f:
    f.write(b"foo")
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls `close()` on contextmanager exit.

class tamp.**TextCompressor**(*f*, *, *window: int = 10*, *literal: int = 8*, *dictionary: bytearray | None = None*, *lazy_matching: bool = False*, *extended: bool = True*, *dictionary_reset: bool = False*, *append: bool = False*)

Compresses text to a file or stream.

Parameters

- **f** (*Union[str, Path, FileLike]*) -- Path/FileHandle/Stream to write compressed data to.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is 7 for storing ascii characters where the most-significant-bit is always 0. Smaller values result in higher compression ratios for no additional computation cost. Valid range: `[5, 8]`.

- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with `initialize_dictionary()`
- **lazy_matching** (*bool*) -- Use roughly 50% more cpu to get 0~2% better compression.
- **append** (*bool*) -- Initialize for appending to an existing stream instead of writing a header. Writes a FLUSH token that, combined with the previous stream's trailing FLUSH, triggers a dictionary reset in the decompressor. Requires `dictionary_reset=True`.

write(*data: str*) → int

Compress `data` to stream.

Parameters
data (*Union[bytes, bytearray]*) -- Data to be compressed.

Returns
Number of compressed bytes written. May be zero when data is filling up internal buffers.

Return type
int

__enter__() → *Compressor*

Use `Compressor` as a context manager.

```
with tamp.Compressor("output.tamp") as f:
    f.write(b"foo")
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls `close()` on contextmanager exit.

close() → int

Flushes all internal buffers and closes the output file or stream, if tamp opened it.

Returns
Number of compressed bytes flushed to disk.

Return type
int

flush(*write_token: bool = True*) → int

Flushes all internal buffers.

This compresses any data remaining in the input buffer, and flushes any remaining data in the output buffer to disk.

Parameters
write_token (*bool*) -- If appropriate, write a `FLUSH` token. Defaults to `True`.

Returns
Number of compressed bytes flushed to disk.

Return type
int

reset_dictionary() → int

Reset the dictionary and internal state, writing a double-FLUSH signal.

This allows the decompressor to re-initialize its dictionary at the corresponding point in the stream. Useful for long streams where the data characteristics change significantly.

Returns
Number of compressed bytes written.

Return type
int

Raises
`ValueError` -- If the compressor was not initialized with `dictionary_reset=True`.

`tamp.compress`(*data: bytes | str, *, window: int = 10, literal: int = 8, dictionary: bytearray | None = None, lazy_matching: bool = False, extended: bool = True*) → bytes

Single-call to compress data.

Parameters

- **data** (*Union[str, bytes]*) -- Data to compress.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is `7` for storing ascii characters where the most-significant-bit is always 0. Valid range: `[5, 8]`.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with `initialize_dictionary()`
- **lazy_matching** (*bool*) -- Use roughly 50% more cpu to get 0~2% better compression.
- **extended** (*bool*) -- Use extended compression format. Defaults to True.

Returns
Compressed data

Return type
bytes

class `tamp.Decompressor`(*f, *, dictionary: bytearray | None = None*)

Decompresses a file or stream of tamp-compressed data.

Can be used as a context manager to automatically handle file opening and closing:

```
with tamp.Decompressor("compressed.tamp") as f:
    decompressed_data = f.read()
```

Parameters

- **f** (*Union[file, str]*) -- File-like object to read compressed bytes from.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with `initialize_dictionary()`

`readinto`(*buf: bytearray*) → int

Decompresses data into provided buffer.

Parameters
`buf` (*bytearray*) -- Buffer to decode data into.

Returns
Number of bytes decompressed into buffer.

Return type
int

read(*size: int = -1*) → bytearray

Decompresses data to bytes.

Parameters
`size` (*int*) -- Maximum number of bytes to return. If a negative value is provided, all data will be returned. Defaults to `-1`.

Returns
Decompressed data.

Return type
bytearray

close()

Closes the input file or stream, if tamp opened it.

__enter__()

Use *Decompressor* as a context manager.

```
with tamp.Decompressor("output.tamp") as f:
    decompressed_data = f.read()
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls *close()* on contextmanager exit.

class tamp.**TextDecompressor**(*f*, *, *dictionary: bytearray | None = None*)

Decompresses a file or stream of tamp-compressed data into text.

Parameters

- **f** (*Union[file, str]*) -- File-like object to read compressed bytes from.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

read(*size: int = -1*) → str

Decompresses data to text.

Parameters
`size` (*int*) -- Maximum number of bytes to return. If a negative value is provided, all data will be returned. Defaults to `-1`.

Returns
Decompressed text.

Return type
str

`__enter__`()

Use `Decompressor` as a context manager.

```
with tamp.Decompressor("output.tamp") as f:
    decompressed_data = f.read()
```

`__exit__`(*exc_type*, *exc_value*, *traceback*)

Calls `close()` on contextmanager exit.

`close`()

Closes the input file or stream, if tamp opened it.

`readinto`(*buf: bytearray*) → int

Decompresses data into provided buffer.

Parameters

buf (`bytearray`) -- Buffer to decode data into.

Returns

Number of bytes decompressed into buffer.

Return type

int

`tamp.`**`decompress`**(*data: bytes*, *, *dictionary: bytearray | None = None*) → bytearray

Single-call to decompress data.

Parameters

- **data** (`bytes`) -- Tamp-compressed data to decompress.
- **dictionary** (`Optional[bytearray]`) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with `initialize_dictionary()`

Returns

Decompressed data.

Return type

bytearray

`tamp.`**`open`**(*f*, *mode='rb'*, ***kwargs*)

Opens a file for compressing/decompressing.

Example usage:

```
with tamp.open("file.tamp", "w") as f:
    # Opens a compressor in text-mode
    f.write("example text")

with tamp.open("file.tamp", "r") as f:
    # Opens a decompressor in text-mode
    assert f.read() == "example text"
```

Parameters

- **f** (`Union[str, Path]`) -- PathLike object to open.
- **mode** (`str`) -- Opening mode. Must be some combination of `{"r", "w", "b"}`.

  - Read-text-mode ("r") will return a *tamp.TextDecompressor*. Read data will be str.
  - Read-binary-mode ("rb") will return a *tamp.Decompressor*. Read data will be bytes.
  - Write-text-mode ("w") will return a *tamp.TextCompressor*. str must be provided to *write()*.
  - Write-binary-mode ("wb") will return a *tamp.Compressor*. bytes must be provided to *write()*.
- **kwargs** -- Passed along to class constructor.

Returns
File-like object for compressing/decompressing.

tamp.**initialize_dictionary**(*source*, *seed=None*, *literal=8*)

Initialize Dictionary.

The character table used for seeding depends on the literal bit width: for literal=7 or 8, common english text and markup characters are used; for literal=5 or 6, common english letters (" etaoinshrdlcumw") downshifted to the target bit width are used instead.

For v1 backwards compatibility, pass literal=8 (the default) when the extended header flag is not set.

Parameters

- **size** (*Union[int, bytearray]*) -- If a bytearray, will populate it with initial data. If an int, will allocate and initialize a bytearray of indicated size.
- **literal** (*int*) -- Number of literal bits (5-8). Selects the appropriate seed character table. Defaults to 8.

Returns
Initialized window dictionary.

Return type
bytearray

exception tamp.**ExcessBitsError**

Provided data has more bits than expected literal bits.

CHAPTER

NINE

C LIBRARY

Tamp provides a C library optimized for low-memory-usage, fast runtime, and small binary footprint. This page describes how to use the provided library.

9.1 Compile-Time Flags

Tamp's C library can be customized via compile-time flags to control features, code size, and performance. Pass these flags to your compiler (e.g., `-DTAMP_STREAM=0`).

Flag	Default	Description
TAMP_COMPRESSOR	1	Include compressor implementation. Set to `0` to exclude compressor code entirely, reducing binary size for decompressor-only builds.
TAMP_DECOMPRESSOR	1	Include decompressor implementation. Set to `0` to exclude decompressor code entirely, reducing binary size for compressor-only builds.
TAMP_ESP32	0	Use ESP32-optimized variant. Avoids bitfields for speed at the cost of slightly higher memory usage. Automatically enabled via Kconfig on ESP-IDF.
TAMP_EXTENDED	1	Default value for extended format support (RLE, extended match encoding). Set to `0` to disable extended support in both compressor and decompressor.
TAMP_EXTENDED_COMPRES	TAMP_EXTEN	Enable extended format compression. Defaults to TAMP_EXTENDED but can be individually overridden for compressor-only or decompressor-only builds.
TAMP_EXTENDED_DECOMPI	TAMP_EXTEN	Enable extended format decompression. Defaults to TAMP_EXTENDED but can be individually overridden for compressor-only or decompressor-only builds.
TAMP_LAZY_MATCHING	0	Enable lazy matching support. When enabled, `TampConf.lazy_matching` becomes available. Improves compression ratio by 0.5-2% at the cost of 50-75% slower compression. Most embedded systems should leave disabled.
TAMP_STREAM	1	Include stream API (`tamp_compress_stream`, `tamp_decompress_stream`). Most users don't need to change this; with `-ffunction-sections` and `--gc-sections` (standard on embedded toolchains), unused stream functions are automatically stripped. Set to `0` to guarantee exclusion.
TAMP_STREAM_FATFS	0	Enable FatFs (ChaN's FAT filesystem) stream handlers. Requires FatFs headers.
TAMP_STREAM_LITTLEFS	0	Enable LittleFS stream handlers. Requires LittleFS headers.
TAMP_STREAM_MEMORY	0	Enable memory buffer stream handlers (`TampMemReader`, `TampMemWriter`). Useful for file-to-memory or memory-to-file operations.
TAMP_STREAM_STDIO	0	Enable stdio (`FILE*`) stream handlers. Works with standard C library, ESP-IDF VFS, and POSIX-compatible systems.
TAMP_STREAM_WORK_BUFI	32	Stack-allocated work buffer size (bytes) for stream API. Split in half for input/output. Larger values reduce I/O callback invocations, improving decompression speed. 256+ bytes recommended when stack permits.
TAMP_USE_MEMSET	1	Use libc `memset`. Set to `0` for environments without libc (e.g. MicroPython native modules). When disabled, uses a volatile byte loop that avoids emitting a `memset` call at the cost of store coalescing.

Example: Minimal decompressor-only build

```
gcc -DTAMP_COMPRESSOR=0 -DTAMP_STREAM=0 -c decompressor.c common.c
```

Example: Full-featured build with LittleFS support

```
gcc -DTAMP_LAZY_MATCHING=1 -DTAMP_STREAM_LITTLEFS=1 -DTAMP_STREAM_WORK_BUFFER_SIZE=256 \
    -Ipath/to/littlefs -c compressor.c decompressor.c common.c
```

9.2 Overview

To use Tamp in your C project, simply copy the contents of `tamp/_c_src` into your project. The contents are broken down as follows (header files described, but you'll also need the c files):

1. `tamp/common.h` - Common functionality needed by both the compressor and decompressor. Must be included.
2. `tamp/compressor.h` - Functions to compress a data stream.
3. `tamp/decompressor.h` - Functions to decompress a data stream.

All header files are well documented. Please refer to the appropriate header file for precise API usage. This document primarily serves as suggestions on how to use the library, and some of the philosophy behind it.

9.3 Compressor

To include the compressor functionality, include the compressor header:

```
#include "tamp/compressor.h"
```

9.3.1 Initialization

All compression is performed using a `TampCompressor` object. The object must first be initialized with `tamp_compressor_init`. The compressor object, a configuration, and a buffer are provided. Tamp performs no internal memory allocations, so a buffer must be provided. Tamp is an LZ-based compression schema, and this buffer is commonly called a "window buffer" or "dictionary". The provided buffer must be at least `(1 << conf.window)` bytes.

```
static unsigned char window_buffer[1024];

TampConf conf = {
   /* Describes the size of the buffer in bits.
   A 10-bit window represents a 1024-byte buffer.
   Must be in range [8, 15], representing [256, 32768] byte windows.
   10 is a good default, balancing compression ratio, memory, and speed. */
   .window = 10,

   /* Number of bits occupied in each plaintext symbol.
   For example, if ASCII text is being encoded, then we could set this
   value to 7 to slightly improve compression ratios.
   Must be in range [5, 8].
   For general use, 8 (the whole byte) is appropriate. */
   .literal = 8,

   /* To improve compression ratios for very short messages, a custom
   buffer initialization could be used.
   For most use-cases, set this to false.*/
```

(continues on next page)

(continued from previous page)

```
    .use_custom_dictionary = false,

    /* Enable lazy matching to slightly improve compression (0.5-2.0%) ratios
    at the cost of 50-75% slower compression.
    Not recommended for most embedded applications.
    Only available when compiled with -DTAMP_LAZY_MATCHING=1. */
    .lazy_matching = false,

    /* Enable extended format (RLE and extended match encoding) for improved
    compression ratios. Default is true in v2. The only downside is
    slightly larger firmware; compression and decompression speed are
    unaffected.
    Only available when compiled with -DTAMP_EXTENDED_COMPRESS=1 (default). */
    .extended = true,

    /* Enable dictionary reset / append support (v2.1.0+).
    Adds a second header byte; old decompressors reject the stream.
    Required for reset_dictionary() and append mode. */
    .dictionary_reset = false,
};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);
```

9.3.2 Compression

Once the `TampCompressor` object is initialized, compression of data can be performed. The compressor is most efficient when provided with larger input buffers, as more context allows better pattern matching. There are two API levels: a higher-level API for most use cases, and a low-level API for fine-grained control.

The higher-level functions handle the internal sink/poll loop:

- `tamp_compressor_compress` - Compresses input until the input is exhausted or the output buffer is full.
- `tamp_compressor_compress_and_flush` - Same as above, but also flushes internal buffers at the end (see *Flushing*).

For most use cases, `tamp_compressor_compress_and_flush` is the right choice.

Low-Level API

The low-level API gives direct control over compression timing, which can be useful for real-time systems:

1. `tamp_compressor_sink` - Copy bytes into the compressor's internal input buffer (up to 16 bytes). This is computationally cheap.
2. `tamp_compressor_poll` - Perform a single compression cycle on the internal buffer, consuming up to 15 bytes (or all 16 with extended format). This is the computationally intensive step, writing 0-3 bytes to the output (up to 5 with extended format, requiring 6 bytes of output buffer space). Compression is most efficient when the internal input buffer is full.

Breaking the operation into two functions allows `tamp_compressor_poll` to be called at a more opportune time in your program.

```c
while(input_size > 0 && output_size > 0){
    {
        // Sink Data into input buffer.
        size_t consumed;
        tamp_compressor_sink(compressor, input, input_size, &consumed);
        input += consumed;
        input_size -= consumed;
    }
    {
        // Perform 1 compression cycle on internal input buffer
        size_t chunk_output_written_size;
        res = tamp_compressor_poll(compressor, output, output_size, &chunk_output_
↪written_size);
        output += chunk_output_written_size;
        output_size -= chunk_output_written_size;
        assert(res == TAMP_OK);
    }
}
```

9.3.3 Flushing

Inside the compressor, there may be up to 16 **bytes** of uncompressed data in the input buffer, and 31 **bits** in the output buffer. This means that the compressed output lags behind the input data stream.

For example, if we compress the 44-long non-null-terminated string `"The quick brown fox jumped over the lazy dog"`, the compressor will produce a 32-long data stream, that decompresses to `"The quick brown fox jumped ov"`. The remaining `"er the lazy dog"` is still in the compressor's internal buffers.

To flush the remaining data, use `tamp_compressor_flush`, which drains the internal input buffer via repeated `tamp_compressor_poll` calls, then flushes the output bit buffer.

```c
tamp_res res;
unsigned char output_buffer[100];
size_t output_written;  // Stores the resulting number of bytes written to output_buffer.

res = tamp_compressor_flush(&compressor, output_buffer, sizeof(output_buffer), &output_
↪written, true);
assert(res == TAMP_OK);
```

The `write_token` parameter controls whether a `FLUSH` token is appended when padding is required:

- Set to `true` if you intend to continue using the compressor. The `FLUSH` token adds 0~2 bytes of overhead.
- Set to `false` when finalizing a stream, to save 0~2 bytes.

When `dictionary_reset` is enabled, `write_token=true` **always** emits a FLUSH token (even when byte-aligned), and closing the stream emits one automatically. This enables the append mode described below.

9.3.4 Dictionary Reset

Note

New in v2.1.0. Requires v2.1.0+ compressor and decompressor.

Dictionary reset allows appending new compressed data to an existing stream without retaining the previous compressor state. After a reset, both compressor and decompressor start fresh with a default dictionary, so no prior window buffer, input buffer, or bit buffer state needs to be persisted.

`tamp_compressor_reset_dictionary` writes a double-FLUSH signal, resets the dictionary and all internal state, then continues with the same configuration. The decompressor handles double-FLUSH resets automatically.

Requires `.dictionary_reset = true` in `TampConf`, which adds a second header byte. Older decompressors reject the stream at the header level. Returns `TAMP_INVALID_CONF` if not set.

```
TampConf conf = {.window = 10, .literal = 8, .dictionary_reset = true};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);

// ... compress some data ...

unsigned char reset_buffer[32];  // Worst case: 27 bytes
size_t reset_written;
res = tamp_compressor_reset_dictionary(&compressor, reset_buffer,
                                       sizeof(reset_buffer), &reset_written);
// Append reset_buffer[0..reset_written] to output, then continue compressing.
```

Append Mode

Set `.append = true` to resume compression on an existing stream without persisting compressor state. Requires:

1. The previous compression stream to have had `dictionary_reset = true`.
2. The new compressor to have the same configuration as the previous compressor (same `window_bits`, `literal`, etc).

Writes a byte-aligned FLUSH token to the internal bit buffer instead of a header. Combined with the previous stream's trailing FLUSH, this forms a double-FLUSH that resets the decompressor's dictionary.

Requires `dictionary_reset = true` and `use_custom_dictionary = false`.

```
// Session 1
TampConf conf = {.window = 10, .literal = 8, .dictionary_reset = true};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);
tamp_compressor_compress_and_flush(&compressor, output, output_size,
                                   &output_written, input, input_size,
                                   &input_consumed, true);
// Save output to storage (stream ends with FLUSH token).

// Session 2
conf.append = true;
TampCompressor compressor2;
```

(continues on next page)

(continued from previous page)

```
tamp_compressor_init(&compressor2, &conf, window_buffer);
tamp_compressor_compress_and_flush(&compressor2, output, output_size,
                                   &output_written, new_input, new_input_size,
                                   &input_consumed, true);
// Append output to storage.
```

9.3.5 Lazy Matching

Lazy matching is a compression optimization that can slightly improve compression ratios at the cost of slower compression speed and increased firmware size. To compile the library with lazy matching support, define the `TAMP_LAZY_MATCHING` macro to 1 during compilation:

```
#define TAMP_LAZY_MATCHING 1
#include "tamp/compressor.h"
```

Alternatively, use the compiler flag `-DTAMP_LAZY_MATCHING=1`.

When compiled with `TAMP_LAZY_MATCHING=1`, the `TampConf.lazy_matching` field becomes available and can be set to enable this feature for individual compressor instances.

9.3.6 Minimizing Output Buffer Size

The following table shows the minimum output buffer size required for each operation. `tamp_compressor_flush` calls `tamp_compressor_poll` until the internal input buffer is exhausted; the flush rows reflect the final output after exhaustion.

Operation	Standard	Extended
`poll`	3 bytes	6 bytes
`flush` (`write_token=false`)	4 bytes	6 bytes
`flush` (`write_token=true`)	5 bytes	7 bytes

If output buffer sizing is a concern, consider using the *Stream API* instead. The stream API handles chunked I/O internally using a small stack-allocated work buffer, eliminating the need to size output buffers manually:

```
// Compile with: -DTAMP_STREAM_STDIO=1

FILE *in = fopen("input.txt", "rb");
FILE *out = fopen("output.tamp", "wb");
unsigned char window[1 << 10];

TampCompressor compressor;
tamp_compressor_init(&compressor, NULL, window);

tamp_compress_stream(
    &compressor,
    tamp_stream_stdio_read, in,
    tamp_stream_stdio_write, out,
    NULL, NULL, NULL, NULL
);
```

(continues on next page)

(continued from previous page)

```
fclose(in);
fclose(out);
```

For users of the buffer-based API, the maximum number of bytes (and thus, the suggested output buffer size) that can be flushed from a compressor with a full internal input buffer via `tamp_compressor_flush` can be calculated as:

$$max_output_size = \left\lceil \frac{16 + \text{window} + 16(1 + \text{literal})}{8} \right\rceil$$

The math for with `write_token=true` is more complicated, but it just so happens that in all valid configuration cases, it requires 1 more byte in the output buffer:

Literal Size (Bits)	Window Size (Bits)	Max Output Size write_token=false (Bytes)	Max Output Size write_token=true (Bytes)
5	8	15	16
5	9-15	16	17
6	8	17	18
6	9-15	18	19
7	8	19	20
7	9-15	20	21
8	8	21	22
8	9-15	22	23

For most applications, `literal=8` and `window=10` offers a good tradeoff, and should have an output buffer size of at least 22 bytes.

9.3.7 Summary

```c
unsigned char window_buffer[1024];
const unsigned char input_string[44] = "The quick brown fox jumped over the lazy dog";
unsigned char output_buffer[64];

TampConf conf = {.window=10, .literal=8};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);

size_t input_consumed_size, output_written_size;
tamp_compressor_compress_and_flush(
     &compressor,
     output_buffer, sizeof(output_buffer), &output_written_size,
     input_string, sizeof(input_string), &input_consumed_size,
     false  // Don't write flush token
);

// Compressed data is now in output_buffer
printf("Compressed size: %zu\n", output_written_size);
```

9.4 Decompressor

The decompressor API is much simpler than the compressor API. To include the decompressor functionality, include the decompressor header:

```
#include "tamp/decompressor.h"
```

9.4.1 Initialization

All decompression is performed using a `TampDecompressor` object. Like `TampCompressor`, this object needs to be configured with a `TampConf` object. Typically, this configuration comes from the Tamp header at the beginning of the compressed data. Use `tamp_decompressor_read_header` to read the header into a `TampConf`:

```
const unsigned char *compressed_data = ...;  // Imagine this points to tamp-compressed␣
↪data.
size_t compressed_data_size = 64;
tamp_res res;
TampConf conf;
size_t compressed_consumed_size;

// This will populate conf.
res = tamp_decompressor_read_header(
    &conf,
    compressed_data, compressed_data_size, &compressed_consumed_size
);
assert(res == TAMP_OK);

compressed_data += compressed_consumed_size;
compressed_data_size -= compressed_consumed_size;

// TODO: actual decompression.
```

Explicitly reading the header is useful if the window-buffer needs to be dynamically allocated. The window-buffer size can be calculated as `(1 << conf.window)`. If a static window buffer is used, then `tamp_decompressor_read_header` doesn't need to be explicitly called. `tamp_decompressor_init` initializes the actual decompressor object, using an optionally supplied `TampConf`. If no `TampConf` is provided, then it will be automatically initialized on first `tamp_decompressor_decompress` call from input header data.

```
TampDecompressor decompressor;
unsigned char window_buffer[1024];
tamp_res res;

// Since no TampConf is provided, the header will automatically be parsed
// in the first tamp_decompressor_decompress call.
// The last parameter (10) indicates the buffer can accommodate up to a 10-bit window.
res = tamp_decompressor_init(&decompressor, NULL, window_buffer, 10);

assert(res == TAMP_OK);
```

9.4.2 Decompression

Data decompression is straightforward:

```
const unsigned char input_data[64]; // Hypothetical input compressed data.
size_t input_consumed_size;

unsigned char output_data[64];  // output decompressed data
size_t output_written_size;

res = tamp_decompressor_decompress(
    &decompressor,
    output_data, sizeof(output_data), &output_written_size,
    input_data, sizeof(input_data), &input_consumed_size
);
// res could be:
//    TAMP_INPUT_EXHAUSTED - All data in input buffer has been consumed.
//    TAMP_OUTPUT_FULL - Output buffer is full.
// In all situations, output_written_size and input_consumed_size are updated.
```

9.5 Callbacks

`tamp_compressor_compress`, `tamp_compressor_compress_and_flush`, and `tamp_decompressor_decompress` each have callback variants (`_cb` suffix) that accept two additional arguments:

- `callback`, a function with signature:

  ```
  int callback(void *user_data, size_t bytes_processed, size_t total_bytes);
  ```

  Where `bytes_processed` is the number of input bytes consumed so far, and `total_bytes` is the `input_size` passed to this call. This allows computing a progress percentage as `bytes_processed / total_bytes`. Return 0 to continue, or non-zero to abort the operation. The return value is truncated to `tamp_res` (`int8_t`); use values in [100, 127] or [-128, -100] for custom codes.

- `void *user_data`, arbitrary data passed along to the callback.

Callbacks are useful for resetting a watchdog, updating a progress bar, etc.

The callback fires once per compression or decompression cycle (i.e., once per encoded or decoded token). Note that `tamp_compressor_compress_cb` only fires the callback when the internal input buffer is full, so trailing input bytes that don't fill the buffer won't trigger a callback. `tamp_compressor_compress_and_flush_cb` addresses this with a final callback after flushing. The *Stream API* also accepts an optional progress callback; it fires once per read-chunk and can abort the stream in the same way (non-zero return).

9.6 Stream API

Tamp provides a high-level stream API for compressing/decompressing between files, memory buffers, or **any custom source/sink**. Tamp is a one-shot compression algorithm: it reads input sequentially and streams output without seeking, making it suitable for non-seekable sources like UART or network sockets. The stream API uses **callbacks** for reading and writing, making it filesystem-agnostic. Built-in handlers are provided for common use cases.

9.6.1 Callback Types

The stream API uses two callback types defined in `common.h`:

```
// Read callback: read up to `size` bytes into `buffer`
// Returns bytes read (0 for EOF), or negative on error
typedef int (*tamp_read_t)(void *handle, unsigned char *buffer, size_t size);

// Write callback: write `size` bytes from `buffer`
// Returns bytes written, or negative on error
typedef int (*tamp_write_t)(void *handle, const unsigned char *buffer, size_t size);
```

A negative return from the read callback is promoted to `TAMP_READ_ERROR`. A return of 0 indicates EOF.

A negative return or incomplete write from the write callback is promoted to `TAMP_WRITE_ERROR`. Chunks are small (see *Work Buffer*), so writing the full amount is expected.

9.6.2 Work Buffer

The stream functions use an internal **stack-allocated** work buffer for intermediate I/O operations. This buffer is split in half internally: one half for reading input, the other for writing output. Larger buffers reduce the number of I/O callback invocations. Compression is generally CPU bound and is unaffected by the work buffer size. Decompression is more IO bound and benefits from a larger work buffer.

The buffer size is controlled by the `TAMP_STREAM_WORK_BUFFER_SIZE` macro, which defaults to `32` bytes. This default is conservative and safe for constrained embedded stacks. The work buffer size **does not** need to be a power of 2.

Benchmark results on the 100MB enwik8 dataset on an Apple M3 MacBook Air:

Work Buffer Size (B)	Compress Time (s)	Decompress Time (s)
32	5.554	0.832
64	5.380	0.671
128	5.314	0.580
256	5.324	0.543
512	5.261	0.507
1024	5.273	0.514

9.6.3 Stream Compression

First initialize a `TampCompressor` with `tamp_compressor_init`, then use `tamp_compress_stream`:

```
TampCompressor compressor;
unsigned char window[1 << 10];

// Initialize the compressor (NULL conf for defaults)
tamp_compressor_init(&compressor, NULL, window);

// Compress from source to destination
tamp_res tamp_compress_stream(
    TampCompressor *compressor,   // Initialized compressor
    tamp_read_t read_cb,          // Callback to read uncompressed input
    void *read_handle,            // Handle passed to read_cb
    tamp_write_t write_cb,        // Callback to write compressed output
    void *write_handle,           // Handle passed to write_cb
    size_t *input_consumed_size,  // Output: total bytes read (may be NULL)
    size_t *output_written_size,  // Output: total bytes written (may be NULL)
    tamp_callback_t callback,     // Progress callback (may be NULL)
    void *user_data               // User data for callback
);
```

The stream progress callback receives `(input_consumed, 0)` — `total_bytes` is 0 because the total input size is not known ahead of time.

9.6.4 Stream Decompression

First initialize a `TampDecompressor` with `tamp_decompressor_init`, then use `tamp_decompress_stream`:

```
TampDecompressor decompressor;
unsigned char window[1 << 10];

// Initialize the decompressor (NULL conf to read from stream header)
tamp_decompressor_init(&decompressor, NULL, window, 10);

// Decompress from source to destination
tamp_res tamp_decompress_stream(
    TampDecompressor *decompressor,   // Initialized decompressor
    tamp_read_t read_cb,              // Callback to read compressed input
    void *read_handle,                // Handle passed to read_cb
    tamp_write_t write_cb,            // Callback to write decompressed output
    void *write_handle,               // Handle passed to write_cb
    size_t *input_consumed_size,      // Output: total bytes read (may be NULL)
    size_t *output_written_size,      // Output: total bytes written (may be NULL)
    tamp_callback_t callback,         // Progress callback (may be NULL)
    void *user_data                   // User data for callback
);
```

The stream progress callback receives `(input_consumed, 0)`, same as compression streams.

9.6.5 Built-in I/O Handlers

Tamp provides built-in read/write handlers for common use cases. The same handlers work for both compression and decompression, and can be mixed (e.g., read from memory, write to file). Enable them by defining the appropriate macro via compiler flags (e.g., `-DTAMP_STREAM_STDIO=1`).

TAMP_STREAM_STDIO

Standard C stdio (`FILE*`). Works with standard C library, ESP-IDF VFS, and any POSIX-compatible system.

```
// Compile with: -DTAMP_STREAM_STDIO=1

FILE *in = fopen("input.txt", "rb");
FILE *out = fopen("output.tamp", "wb");
unsigned char window[1 << 10];

TampCompressor compressor;
tamp_compressor_init(&compressor, NULL, window);

tamp_compress_stream(
    &compressor,             // initialized compressor
    tamp_stream_stdio_read,  // read callback
    in,                      // read handle
    tamp_stream_stdio_write, // write callback
    out,                     // write handle
    NULL,                    // input_consumed_size (optional)
    NULL,                    // output_written_size (optional)
    NULL,                    // progress callback (optional)
    NULL                     // callback user_data (optional)
);

fclose(in);
fclose(out);
```

TAMP_STREAM_MEMORY

Memory buffer handlers, typically used for file-to-memory or memory-to-file operations. For memory-to-memory operations, use `tamp_compressor_compress_and_flush` or `tamp_decompressor_decompress` directly.

```
// Compile with: -DTAMP_STREAM_MEMORY=1 -DTAMP_STREAM_STDIO=1

const unsigned char input_data[] = "Data to compress...";
unsigned char window[1 << 10];

// Setup memory reader for input
TampMemReader reader = {
    .data = input_data,
    .size = sizeof(input_data),
    .pos = 0
};

// Initialize compressor
```

(continues on next page)

(continued from previous page)

```
TampCompressor compressor;
tamp_compressor_init(&compressor, NULL, window);

// Compress from memory to file
FILE *out = fopen("output.tamp", "wb");
tamp_compress_stream(
    &compressor,            // initialized compressor
    tamp_stream_mem_read,   // read callback (read plaintext from memory)
    &reader,                // read handle
    tamp_stream_stdio_write, // write callback (write compressed data to file)
    out,                    // write handle
    NULL,                   // input_consumed_size (optional)
    NULL,                   // output_written_size (optional)
    NULL,                   // progress callback (optional)
    NULL                    // callback user_data (optional)
);
fclose(out);
```

TAMP_STREAM_LITTLEFS

LittleFS is a fail-safe filesystem designed for embedded systems. Unlike stdio where a single `FILE*` handle is sufficient, LittleFS requires both the filesystem object (`lfs_t`) and the file handle (`lfs_file_t`) for I/O operations. Tamp provides a `TampLfsFile` struct to bundle these together into a single handle.

```
// Compile with: -DTAMP_STREAM_LITTLEFS=1 -Ipath/to/littlefs

lfs_t lfs;
lfs_file_t in_file, out_file;
// Assumes lfs_mount(&lfs, &cfg) has already been called
lfs_file_open(&lfs, &in_file, "data.tamp", LFS_O_RDONLY);
lfs_file_open(&lfs, &out_file, "data.bin", LFS_O_WRONLY | LFS_O_CREAT);

// Bundle filesystem and file handle together
TampLfsFile in_handle = { .lfs = &lfs, .file = &in_file };
TampLfsFile out_handle = { .lfs = &lfs, .file = &out_file };
unsigned char window[1 << 10];

// Initialize decompressor (NULL conf reads from stream header)
TampDecompressor decompressor;
tamp_decompressor_init(&decompressor, NULL, window, 10);

tamp_decompress_stream(
    &decompressor,          // initialized decompressor
    tamp_stream_lfs_read,   // read callback (read compressed data from file)
    &in_handle,             // read handle
    tamp_stream_lfs_write,  // write callback (write decompressed data to file)
    &out_handle,            // write handle
    NULL,                   // input_consumed_size (optional)
    NULL,                   // output_written_size (optional)
    NULL,                   // progress callback (optional)
    NULL                    // callback user_data (optional)
```

(continues on next page)

(continued from previous page)

```
);

lfs_file_close(&lfs, &in_file);
lfs_file_close(&lfs, &out_file);
```

TAMP_STREAM_FATFS

FatFs (ChaN's FAT filesystem) is commonly used for SD cards and USB mass storage on embedded systems. Unlike LittleFS, FatFs file handles (FIL) are self-contained and can be passed directly to the callbacks.

```
// Compile with: -DTAMP_STREAM_FATFS=1 -Ipath/to/fatfs

FIL in_file, out_file;
f_open(&in_file, "data.tamp", FA_READ);
f_open(&out_file, "data.bin", FA_WRITE | FA_CREATE_ALWAYS);
unsigned char window[1 << 10];

// Initialize decompressor (NULL conf reads from stream header)
TampDecompressor decompressor;
tamp_decompressor_init(&decompressor, NULL, window, 10);

tamp_decompress_stream(
    &decompressor,           // initialized decompressor
    tamp_stream_fatfs_read,  // read callback
    &in_file,                // read handle
    tamp_stream_fatfs_write, // write callback
    &out_file,               // write handle
    NULL,                    // input_consumed_size (optional)
    NULL,                    // output_written_size (optional)
    NULL,                    // progress callback (optional)
    NULL                     // callback user_data (optional)
);

f_close(&in_file);
f_close(&out_file);
```

CHAPTER

TEN

JAVASCRIPT/TYPESCRIPT API

The Tamp WebAssembly package provides JavaScript and TypeScript bindings for the Tamp compression library.

10.1 Installation

```
npm install @brianpugh/tamp
```

10.2 Basic Usage

10.2.1 Simple Compression

Compress/decompress in-memory in a single operation:

```
import { compress, decompress, compressText, decompressText } from '@brianpugh/tamp';

// Compress binary data
const originalData = new TextEncoder().encode('Hello, World!');
const compressed = await compress(originalData);
const decompressed = await decompress(compressed);

// Compress text directly
const compressedText = await compressText('Hello, World!');
const restoredText = await decompressText(compressedText);
```

10.2.2 Configuration Options

Customize compression behavior with options:

```
const options = {
  // Describes the size of the decompression buffer in bits.
  // A 10-bit window represents a 1024-byte buffer.
  // Must be in range [8, 15], representing [256, 32768] byte windows.
  window: 12,

  // Number of bits occupied in each plaintext symbol.
  // For example, if ASCII text is being encoded, then we could set this
```

(continues on next page)

(continued from previous page)

```
  // value to 7 to slightly improve compression ratios.
  // Must be in range [5, 8].
  // For general use, 8 (the whole byte) is appropriate.
  literal: 7,

  // Enable extended format (RLE, extended match) for better compression ratios.
  // The extended format provides better compression for typical data at the
  // cost of slightly more complex encoding.
  // Default: true
  extended: true,

  // Enable lazy matching to slightly improve compression (0.5-2.0%) ratios
  // at the cost of 50-75% slower compression.
  // Most embedded systems will **not** want to use this feature and disable it.
  lazy_matching: true,

  // To improve compression ratios for very short messages, a custom
  // buffer initialization could be used.
  // For most use-cases, set this to null.
  dictionary: null
};

const compressed = await compress(data, options);
const decompressed = await decompress(compressed, options);
```

10.2.3 Streaming for Large Data

Use streaming compression for processing large files or data that does not fit in memory:

```
import { TampCompressor, TampDecompressor, using } from '@brianpugh/tamp';

// Automatic resource management (recommended)
const compressedChunks = [];
await using(new TampCompressor(options), async (compressor) => {
  for (const chunk of dataChunks) {
    const compressedChunk = await compressor.compress(chunk);
    if (compressedChunk.length > 0) {
      compressedChunks.push(compressedChunk);
    }
  }
  // Flush remaining data
  const finalChunk = await compressor.flush();
  if (finalChunk.length > 0) {
    compressedChunks.push(finalChunk);
  }
});

// Dictionary reset mid-stream; useful if we need to append
// data to a compression stream.
await using(new TampCompressor({ ...options, dictionary_reset: true }), async
↪(compressor) => {
```

(continues on next page)

(continued from previous page)

```
  // Compress first segment
  compressedChunks.push(await compressor.compress(segment1));

  // Reset dictionary
  const resetBytes = await compressor.resetDictionary();
  if (resetBytes.length > 0) {
    compressedChunks.push(resetBytes);
  }

  // Continue compressing with a fresh dictionary
  compressedChunks.push(await compressor.compress(segment2));
  compressedChunks.push(await compressor.flush());
});
```

10.2.4 Web Streams Integration

Use with the Web Streams API for seamless integration with modern web applications:

```
import { TampCompressionStream, TampDecompressionStream } from '@brianpugh/tamp/streams';

// Compress a file stream
const fileStream = file.stream();
const compressionStream = new TampCompressionStream(options);
const compressedStream = fileStream.pipeThrough(compressionStream);

// Save compressed data
const response = new Response(compressedStream);
const compressedBlob = await response.blob();

// Or chain compression and decompression
const decompressionStream = new TampDecompressionStream(options);
const roundTripStream = fileStream
  .pipeThrough(compressionStream)
  .pipeThrough(decompressionStream);
```

10.3 Error Handling

The library throws specific error types:

- `TampError` - Base error class
- `CompressionError` - Compression operation errors
- `DecompressionError` - Decompression operation errors
- `ExcessBitsError` - Data exceeds literal size limits

10.4 Custom Configuration

Configure compression parameters by passing in options:

```
const options = {
  window: 12,           // Larger window for (usually) better compression
  literal: 7,           // ASCII text only requires 7 bits.
  extended: true,       // Enable extended format (RLE, extended match)
  lazy_matching: true   // Better compression ratios; slower to compress
};

const compressed = await compress(data, options);
const decompressed = await decompress(compressed, options);
```

10.5 Progress Callbacks

Monitor compression progress for large files using progress callbacks. Progress callbacks receive a `progressInfo` object with detailed compression metrics:

```
// Basic progress monitoring
const progressCallback = (progressInfo) => {
  console.log(`${progressInfo.percent.toFixed(1)}% complete`);
  console.log(`Speed: ${(progressInfo.bytesPerSecond / 1024).toFixed(1)} KB/s`);
  console.log(`ETA: ${progressInfo.estimatedTimeRemaining.toFixed(1)}s`);
};

const compressed = await compress(largeData, options, progressCallback);

// Conditional abortion example
const abortingCallback = (progressInfo) => {
  console.log(`Progress: ${progressInfo.percent.toFixed(1)}%`);

  // Throw error to abort compression after 50%
  if (progressInfo.percent > 50) {
    throw new Error('Compression cancelled by user');
  }
};

try {
  const compressed = await compress(data, options, abortingCallback);
} catch (error) {
  console.log('Compression was aborted by callback');
}
```

Progress Info Object Fields:

```
interface ProgressInfo {
  bytesProcessed: number;        // Number of input bytes processed so far
  totalBytes: number;            // Total number of input bytes
  percent: number;               // Completion percentage (0-100)
  bytesPerSecond: number;        // Processing speed in bytes/second
```

(continues on next page)

(continued from previous page)

```
  estimatedTimeRemaining: number; // Estimated seconds remaining
  chunksProcessed: number;        // Number of chunks processed
  elapsedTime: number;            // Total elapsed time in seconds
  chunkSize: number;              // Size of current chunk being processed
  outputSize: number;             // Size of output written for current chunk
}
```

The decompressor does not have a progress callback since the operation is usually very very fast.

10.6 Utility Functions

10.6.1 WebAssembly Initialization

The WebAssembly module initializes automatically, but you can preload it explicitly:

```
import { initialize } from '@brianpugh/tamp';

// Preload WebAssembly module (optional)
await initialize();
```

10.6.2 Dictionary Utilities

Create and manage custom dictionaries for improved compression:

```
import { initializeDictionary, computeMinPatternSize } from '@brianpugh/tamp';

// Initialize a dictionary buffer with default values
const dictionary = await initializeDictionary(1024); // Must be power of 2
const dict6bit = await initializeDictionary(1024, 6); // Tuned for 6-bit literals

// Compute minimum pattern size for given parameters
const minPatternSize = await computeMinPatternSize(12, 8); // window=12, literal=8
```

10.6.3 Stream Helpers

Additional utilities for working with streams:

```
import {
  compressStream,
  decompressStream,
  createReadableStream,
  collectStream
} from '@brianpugh/tamp';

// Convert Uint8Array to ReadableStream
const data = new TextEncoder().encode('Hello, World!');
const readableStream = createReadableStream(data, 1024); // 1024 byte chunks
```

(continues on next page)

(continued from previous page)

```
// Compress a readable stream
const compressedStream = compressStream(readableStream, options);

// Decompress a readable stream
const decompressedStream = decompressStream(compressedStream, options);

// Collect stream back to Uint8Array
const result = await collectStream(decompressedStream);
```

10.7 Node.js and Browser Support

The package supports both Node.js (>= 14.0.0) and modern browsers with WebAssembly support. It provides CommonJS and ES Module builds.

```
// ES Modules
import { compress } from '@brianpugh/tamp';

// CommonJS
const { compress } = require('@brianpugh/tamp');
```

CHAPTER

ELEVEN

RUST

Tamp has **unofficial** Rust bindings available through the *tamp-rs* crate, which provides both safe Rust bindings and low-level FFI access to the C library.

Over time, these bindings **may** become integrated with the official Tamp repo.

11.1 Documentation

- tamp crate documentation
- tamp-sys FFI documentation
- GitHub repository

11.2 Installation

Add the following to your `Cargo.toml`:

```
[dependencies]
tamp = "*"  # Use latest version
```

Or using cargo from the command line:

```
cargo add tamp
```

11.3 Example Usage

Here's a simple example showing how to compress and decompress data using the Rust bindings:

```
use tamp::{Compressor, Decompressor, Config};

// Use 1K buffer size (must match 2^window_bits)
const WINDOW_SIZE: usize = 1 << 10;

fn main() -> Result<(), tamp::Error> {
    // Original data to compress - repetitive text compresses better
    let input = b"I scream, you scream, we all scream for ice cream!";

```

(continues on next page)

(continued from previous page)

```
    // Create configuration with window_bits=10 (2^10 = 1024)
    let config = Config {
        window_bits: 10,      // 2^10 = 1024 bytes
        literal_bits: 7,      // Use 7-bit literals for text (ASCII) compression
        ..Default::default()
    };

    // Instantiate the Compressor + Buffers
    let mut compressor: Compressor<WINDOW_SIZE> = Compressor::new(config.clone())?;
    let mut output_buffer = vec![0u8; 1024];
    let mut compressed = Vec::new();

    // Compress chunk and flush (without end token)
    let (_, written) = compressor.compress_chunk(input, &mut output_buffer)?;
    compressed.extend_from_slice(&output_buffer[..written]);
    let written = compressor.flush(&mut output_buffer, false)?;
    compressed.extend_from_slice(&output_buffer[..written]);

    println!("Compressed {} bytes to {} bytes", input.len(), compressed.len());

    // Decompress the data - create decompressor from header
    let (mut decompressor, header_size): (Decompressor<WINDOW_SIZE>, usize) =
        Decompressor::from_header(&compressed)?;

    let mut decompressed = vec![0u8; 1024];
    let (_, written) = decompressor.decompress_chunk(
        &compressed[header_size..],  // Skip header bytes
        &mut decompressed
    )?;

    println!("Decompressed back to {} bytes", written);
    assert_eq!(input, &decompressed[..written]);

    Ok(())
}
```

CHAPTER

TWELVE

MIGRATING TO V2

Tamp v2 introduces the extended compression format (RLE and extended match encoding) and several API improvements. This page covers what changed and how to update your code.

Important

The extended format (`extended=True`) is **enabled by default** in v2. Data compressed with v2 defaults **cannot be decompressed by tamp <2.0.0**. Older decompressors will safely return an error or raise an exception (they validated that the now-used header bit was reserved).

If you need to produce data readable by older tamp versions, explicitly disable the extended format:

- C: `TampConf conf = {... .extended = 0};`
- Python: `tamp.Compressor(f, extended=False)`
- JavaScript: `compress(data, { extended:  false })`

12.1 Format Changes

The v2 format is fully backwards-compatible for decompression: v2 decompressors handle both v1 and v2 streams automatically. No changes are needed to decompress existing v1 data.

12.1.1 Extended Format

When `extended=True` (the default), two new token types improve compression:

- **RLE (Run-Length Encoding)**: Efficiently encodes runs of repeated bytes. Up to 241 consecutive identical bytes per token.
- **Extended Match**: Allows pattern matches much longer than the v1 maximum of `min_pattern_size + 13`, up to `min_pattern_size + 131`.

See *Specification* for encoding details.

12.1.2 Header Bit [1]

Bit [1] of the stream header, previously reserved (always 0), now indicates whether the stream uses the extended format. v2 decompressors use this bit to transparently handle both formats. Existing v1 decompressors validated that this bit was 0, so they will safely return an error or raise an exception when encountering a v2 stream rather than silently producing corrupt output.

12.2 C Library

12.2.1 Enabling Extended Format

To produce v2 output, set `extended = 1` in the configuration. Omitting `.extended` defaults to 0 (v1 format):

```
// v1 output (default, unchanged from before).
TampConf conf = {.window = 10, .literal = 8};

// v2 extended format (new default).
TampConf conf = {.window = 10, .literal = 8, .extended = 1};
```

12.2.2 Callback Semantics

The `tamp_callback_t` progress callback now consistently passes **input bytes consumed** as `bytes_processed` across all API functions.

Previously, the callback behavior was inconsistent:

```
v1 behavior:
  compress_cb:       (output_written, total_input)   -- mixed units
  compress_stream:   (input_consumed, 0)
  decompress_cb:     (output_written, input_size)    -- mixed units
  decompress_stream: (output_written, 0)

v2 behavior (all functions):
  bytes_processed = input bytes consumed so far
  total_bytes     = total input size, or 0 for stream API
```

This means `bytes_processed / total_bytes` now gives a meaningful progress percentage for the non-stream API. Callbacks used only for watchdog resets are unaffected.

If your callback relied on `bytes_processed` being output bytes written, update it to expect input bytes consumed instead.

12.2.3 Struct Layout Changes

`TampConf` has a new `extended` bitfield inserted before `lazy_matching`, which changes the bit layout. Code that manipulates `TampConf` as raw bits must be updated.

`TampCompressor` and `TampDecompressor` have been reorganized and have new fields for extended format state. `sizeof()` of both structs has increased. Code that accesses struct internals directly (rather than through the API) must be updated.

12.2.4 New Compile-Time Flags

The `TAMP_EXTENDED` family of flags controls extended format support:

- `TAMP_EXTENDED` (default 1): Master switch.
- `TAMP_EXTENDED_COMPRESS` (default `TAMP_EXTENDED`): Compressor-only override.
- `TAMP_EXTENDED_DECOMPRESS` (default `TAMP_EXTENDED`): Decompressor-only override.

No existing flags changed behavior. Setting `-DTAMP_EXTENDED=0` disables extended format entirely, producing a v1-only build.

12.2.5 Dictionary Initialization

`tamp_initialize_dictionary` now takes a `literal` parameter to select a seed character table appropriate for the configured literal bit width. For v1 backwards compatibility, callers should pass `literal=8` when `extended` is not set. See *Dictionary Initialization* for details.

12.3 Installation

The CLI is now an optional extra. If you use the `tamp` command line tool, update your install command:

```
pip install tamp[cli]
```

The core library (`pip install tamp`) no longer pulls in CLI dependencies.

12.4 CLI

The new `--extended` flag defaults to enabled. To produce v1 output:

```
tamp compress --no-extended input.txt output.tamp
```

Decompression handles both v1 and v2 data automatically:

```
tamp decompress output.tamp restored.txt
```

12.5 Python

The new `extended` parameter defaults to `True`:

```
# Produces v2 output by default.
compressor = tamp.Compressor(f)

# Explicitly produce v1 output for older decompressors.
compressor = tamp.Compressor(f, extended=False)
```

12.6 JavaScript/WASM

The new `extended` option defaults to `true`:

```
// Produces v2 output by default.
const compressed = await compress(data);

// Explicitly produce v1 output for older decompressors.
const compressed = await compress(data, { extended: false });
```

INDEX

Symbols

__enter__() (*tamp.Compressor method*), 30
__enter__() (*tamp.Decompressor method*), 33
__enter__() (*tamp.TextCompressor method*), 31
__enter__() (*tamp.TextDecompressor method*), 33
__exit__() (*tamp.Compressor method*), 30
__exit__() (*tamp.Decompressor method*), 33
__exit__() (*tamp.TextCompressor method*), 31
__exit__() (*tamp.TextDecompressor method*), 34

B

built-in function
    tamp.open(), 34

C

close() (*tamp.Compressor method*), 30
close() (*tamp.Decompressor method*), 33
close() (*tamp.TextCompressor method*), 31
close() (*tamp.TextDecompressor method*), 34
compress() (*in module tamp*), 32
Compressor (*class in tamp*), 29

D

decompress() (*in module tamp*), 34
Decompressor (*class in tamp*), 32

E

ExcessBitsError, 35

F

flush() (*tamp.Compressor method*), 29
flush() (*tamp.TextCompressor method*), 31

I

initialize_dictionary() (*in module tamp*), 35

R

read() (*tamp.Decompressor method*), 33
read() (*tamp.TextDecompressor method*), 33
readinto() (*tamp.Decompressor method*), 32
readinto() (*tamp.TextDecompressor method*), 34
reset_dictionary() (*tamp.Compressor method*), 30
reset_dictionary() (*tamp.TextCompressor method*), 31

T

tamp.open()
    built-in function, 34
TextCompressor (*class in tamp*), 30
TextDecompressor (*class in tamp*), 33

W

write() (*tamp.Compressor method*), 29
write() (*tamp.TextCompressor method*), 31