

nanocode

nc16fc

Manual for the nanocode fast compression library

Contents

1	Copyright	3
2	List of files	3
3	Overview	4
4	How to use this manual	5
5	Data formats and operating systems supported, and algorithms available	5
6	Platform and runtime characteristics	7
6.1	System requirements	7
6.2	Memory	7
6.3	System calls	7
6.4	Security	8
6.5	Error codes	8
6.6	Threading and configuration lifetime	8
7	Using the nanocode compressors	8
7.1	Compressor / decompressor calls	9
7.1.1	Automatic ORCA geometry detection	10
7.2	Block header and checksum	12
7.3	Compiling and linking a program in Linux using the library	13
7.4	Compiling and linking a program in Windows using the library	14
7.5	Running the supplied programs in Linux	15
7.6	Running the supplied programs in Windows	15
7.7	Accessing the library from Python	16
8	Your data always stays readable	16
9	Licence notice	17
10	Support contact	17
11	Annex A: Included formats and representative benchmarks	18
12	Annex B: Minimal example program	21
13	Annex C: Example compressor	23
14	Annex D: Example decompressor	26

1 Copyright

Copyright © 2026 nanocode GmbH. Use, copying and distribution of the proprietary nc16fc software are governed by the applicable licence agreement. Third-party components are governed by their respective included licence terms. Ownership of the proprietary nc16fc software remains with nanocode GmbH. This manual describes the accompanying library version.

2 List of files

The package comes with the following files:

- `Makefile` : builds the supplied C examples against the accompanying library
- `README.txt` : package quick start and delivery information
- `CHANGES.txt` : customer release notes
- `nc16fc.h` : header file for the nc16fc library
- `nc16fc_manual.pdf` : the file you are reading right now
- `nc16fc_datasheet.pdf` : the data sheet
- `nc16fc_benchmark.c` : benchmark program source; tests the configured algorithm/data combinations listed by `--list`, which can be a subset of the supplied formats
- `nc16fc_compress.c` : compression program source
- `nc16fc_decompress.c` : decompression program source
- `nc16fc_minimal_example.c` : minimal example program source
- `nc16fcpy.py` : Python module that allows accessing the nc16fc C library
- `nc16fc_compress.py` : example Python script that compresses a file (output files are equal to output files of `nc16fc_compress.c`)
- `nc16fc_decompress.py` : example Python script that decompresses a file compressed with `nc16fc_compress.py/nc16fc_compress.c`
- `THIRD_PARTY_NOTICES_AND_LICENSES.txt` : notices and licence texts of the third-party components
- `PACKAGE.json` : package identity, configuration limits and per-file SHA-256 checksums
- `nc16fc_syscalls.txt` : library syscall report

`DELIVERY.json` accompanies the delivery archives and records their identities and SHA-256 checksums. Keep it with the matching archives. Checksums detect changes; they do not constitute a digital signature. Image converters and a separate file comparison program are not part of customer deliveries.

The C example programs are supplied as sources only. Build them with the Makefile before use. The C/Python file examples include their own safe file handling.

For Linux, the static library `nc16fc.a` and the dynamic library `nc16fc.so` are delivered. Executable file names have no extension.

For Windows, `nc16fc.lib` is the import library for the dynamic library `nc16fc.dll`; it is not a standalone static codec library. Executable file names have the `.exe` extension.

3 Overview

nanocode's compressors are developed for lossless compression of high-volume video data streams or transient recording data streams. Key design features of nanocode's compressors include:

- extremely high speed of compression and de-compression
- low resource needs, e.g. for mobile applications, enabling FPGA implementation in addition to the software versions described here
- good compression ratio
- flexible and simple use, both in Windows and Linux-based environments

Typical applications include recording of automotive streams of video images and other sensor data, for development and test of algorithms for autonomous driving.

nanocode's lossless compressors typically allow increasing data recording rates by a factor 2 – in case of disk writing speeds limiting the rate – or increasing data volume by a factor 2 – in case of disk volume limitations. The performance of nanocode's compressors is based both on the use of the AVX instruction set and on efficient low-footprint code.

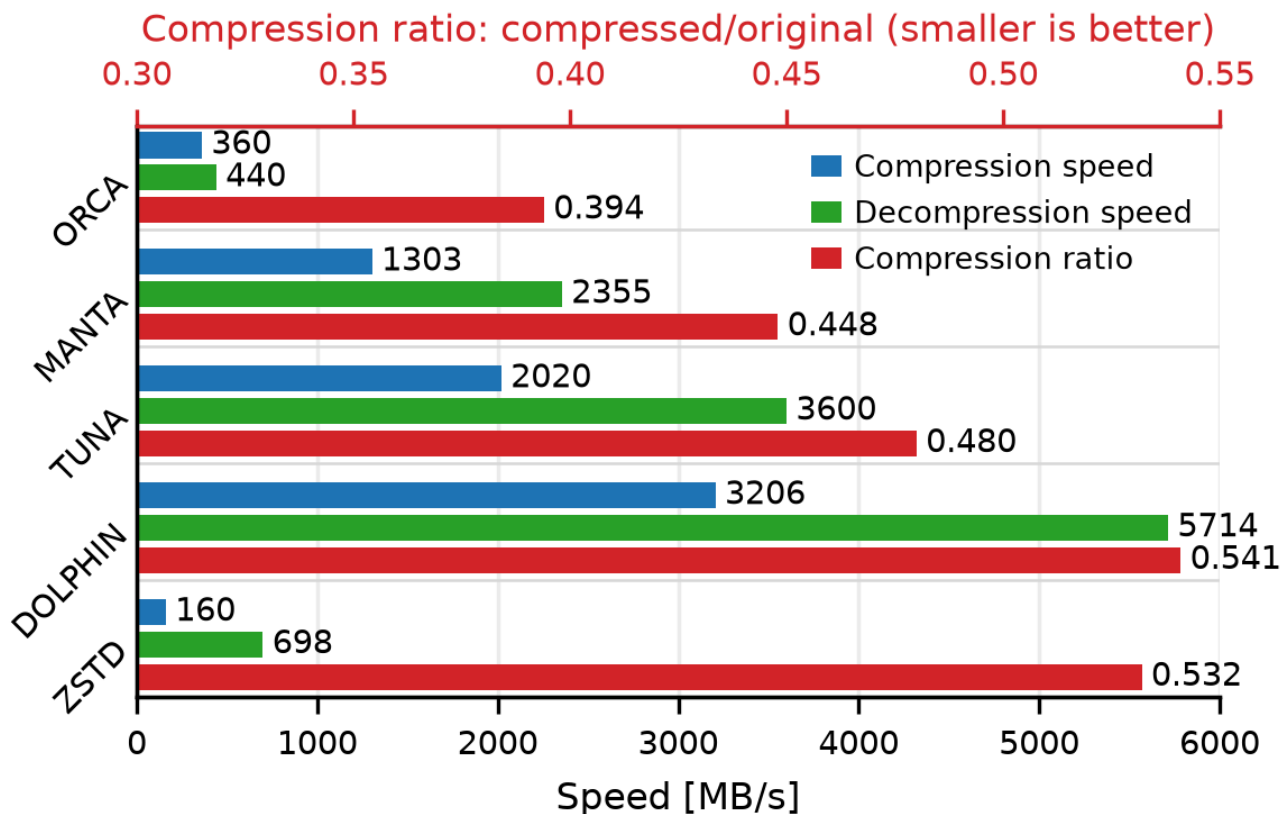


Figure 1: Representative single-thread compression performance

The blue bars show compression speed and the green bars decompression speed, in MB/s on the bottom axis. Red bars show compression ratio on the top axis (compressed size divided by uncompressed size; lower is better). The plot is shared with the datasheet. Results depend on the input data and processor; ZSTD is a comparison baseline, not an additional customer compressor.

nanocode's lossless compressors are available for a range of video formats and sizes of the data frame used for compression (the latter is relevant in particular for parallel FPGA implementations). In addition to the compressor versions described in this manual, additional versions are available upon request. Lossy compressors with adjustable compression ratio in development, again optimized towards maximum speed and low resource footprint.

4 How to use this manual

This manual describes the use of the nanocode lossless video stream compressors, and in particular of the demo programs provided. The description of the tool assumes that the reader is familiar with basics of video formats and video compressors, and of the basic use of the operating systems. The description applies to standard video formats; custom formats – e.g. embedding additional frame information – may require adaption of the compressors; contact nanocode for information.

5 Data formats and operating systems supported, and algorithms available

The software compressors described in this manual support:

- any pixel bit depth up to 16 bit
- packed 12-bit MIPI CSI-2 Bayer images or sample values stored in 8 or 16 bits
- Bayer patterns (RGGB, RYYCy, CCCR, ...) and grayscale images
- interleaved 8-bit input with one to four channels, unsigned or signed
- interleaved 16-bit little-endian input with one to four channels, unsigned or signed; big-endian with one or two
- packed YUV 4:2:2 UYVY; NV12/NV21 through separate Y and interleaved UV/VU planes
- 16-bit data centred on or close to zero, e.g. radar raw data (real samples or I/Q pairs), with one to four channels
- variable image sizes within the limits of the selected format and API

The table shows product support; Annex A lists the formats enabled in this particular package:

Input	MANTA	DOLPHIN	TUNA	ORCA
8 bit, 1–4 channels, unsigned or signed	x	x		x
12 bit CSI-2 packed Bayer	x	x	x	x
16 bit little endian, 1–2 ch., unsigned or signed	x	x	x	x
16 bit little endian, 3–4 ch., unsigned or signed	x	x		x
16 bit big endian, 1–2 ch.	x	x	x	x
YUV 4:2:2 UYVY	x	x		x
16 bit near zero (e.g. radar), 1–4 ch., unsigned or signed		x		

The compressors expect the selected sample packing and image geometry. A 16-bit format preserves all 16 bits, even when the sensor uses fewer. Container headers and metadata between frames must be handled separately so that frame boundaries remain aligned. For NV12/NV21, process the Y plane with an 8BIT format and the UV/VU plane with 8BIT_2CHANNELS at half the luma width and height; there is no combined NV12/NV21 compressor. For NV16 (4:2:2), the UV plane has the full luma height.

Annex A lists the formats included in this package.

Speed: every compressor uses AVX2 where the processor offers it and a fully compatible standard implementation where it does not. The choice is made when the compressor runs, so one name serves both; the two write the same bytes, so a file written on one machine is readable on the other. An FPGA core is available for TUNA, others can be requested.

Algorithms: MANTA balances compression ratio and speed. DOLPHIN prioritizes throughput and a small memory footprint. TUNA supports compatible host and FPGA integration. ORCA targets a high compression ratio for image data. Each operates on individual images; multi-frame compression is not supported.

Operating systems: Currently, Windows- and Linux-based environments are supported.

Mode of use: nanocode's software compressors can be used in three different modes:

- Linked statically into a compiled executable
- Linked dynamically into an executable via a DLL or SO
- From Python

Detailed instructions on use are provided below, together with examples.

Not all combinations of formats and algorithms are currently implemented, but can be provided upon request.

6 Platform and runtime characteristics

6.1 System requirements

Linux/Windows packages target 64-bit Intel/AMD x86-64. AVX2 is required for high compression and decompression speeds; compatible scalar implementations are selected on older CPUs. Faster backends are automatically selected via runtime dispatch when supported by the running CPU. DOLPHIN and TUNA acceleration additionally requires BMI1 and POPCNT.

The library is built without libc and has no dynamic dependency on glibc or musl. It is intended for integration on x86-64 Linux distributions, including Debian/Ubuntu, RHEL-compatible systems and Alpine. The Linux ABI and, for a time-limited build, the required `clock_gettime` and `clock_nanosleep` system calls must be available.

Building the C examples requires a C compiler: GCC or Clang on Linux, or Visual Studio with the C++ development tools and Windows SDK on Windows. The supplied GNU Makefile can also be used with MinGW-w64 in MSYS2. The resulting programs use the C library and compiler runtime of that build environment. Python examples require Python 3 and the matching shared library. No precompiled example programs are supplied.

6.2 Memory

The library allocates no memory. The application provides the memory of each configuration: `nc16fc_memory_size(format, nx, ny)` returns how many bytes `nc16fc_init` needs for a format and image size, and the application passes that memory to `nc16fc_init` – from `malloc`, a static array or a memory pool of its operating system. Any alignment is accepted; the size includes room for the library's own alignment. The memory must stay untouched until `nc16fc_deinit`; after that it may be freed or reused.

The need depends on the compressor: a few kilobytes for DOLPHIN and TUNA, about 1 to 5 MiB for MANTA, about 9 to 10.4 MiB for ORCA, independent of the image size.

`nc16fc_memory_size(NC16FC_ANY_FORMAT, 0, 0)` returns the largest need of all formats of the package, for memory that is reused for several formats. Nothing large is placed on the application's stack: the library's product compressors need at most 48 KiB of stack memory.

At most `max_configs` configurations, recorded in `PACKAGE.json`, can be in use at the same time; beyond that `nc16fc_init` returns `ERROR_CONFIGURATIONS_EXHAUSTED`. The library keeps only this small list of the configurations in use.

6.3 System calls

A system call requests a service from the operating-system kernel. `nc16fc_syscalls.txt` lists the library's imported functions, direct syscall instructions and calls observed during a defined test of the packaged formats. Its scope is `nc16fc.so` and `nc16fc.a`, not the

application that invokes them. An unlimited build of this libc-free library makes no system calls. A trial reads the clock to check whether its licence has expired. After expiry, decoder rate limiting also requires monotonic clock reads and timed sleeps, using `clock_gettime` and `clock_nanosleep` in the libc-free build.

The timing helpers `nc16fc_tic()` and `nc16fc_toc(t)` in `nc16fc.h` read the application's clock. File and timing helpers in the header execute in the calling application.

6.4 Security

The library does not open network connections or files, start threads, or allocate heap memory.

The library writes no output and never ends the application. In an unlimited Linux build without libc, it does not even make system calls.

The syscall report helps review dependencies and sandbox permissions. It is not a security certification or proof that every possible execution path and input is safe. Callers must supply valid buffers with the declared sizes, check return values and keep input and output buffers separate.

6.5 Error codes

Every function that can fail returns a negative error code. `nc16fc.h` defines them in `enum Error_Type` (for example `ERROR_COMPRESSED_BUFFER_TOO_SMALL`); their values do not change between versions, and new codes are only added at the end. `nc16fc_get_error(config)` returns the name of the last error of a configuration.

6.6 Threading and configuration lifetime

A configuration may be used from any thread, but never from two threads at the same time. Applications can use a separate configuration per worker thread. A call that finds its configuration busy with another thread returns `ERROR_STILL_PROCESSING` immediately and changes nothing, not even the text returned by `nc16fc_get_error`. The number of configurations that can be in use at the same time is fixed per package (`max_configs`). `nc16fc_init` and `nc16fc_deinit` may be called from several threads at once. `nc16fc_deinit` returns `ERROR_STILL_PROCESSING` while another thread is in a call with that configuration; give a configuration up only after its operations have finished, do not use it afterwards, and free its memory only after `nc16fc_deinit` returned 0. The library starts no threads of its own.

7 Using the nanocode compressors

The following sections illustrate:

- the basic calls to the compressor library

- how to compile a program using the compressor library on a Linux system
- how to compile a program using the compressor library on a Windows system
- how to build and run the C example programs on Linux or Windows

The use of the compressors is illustrated using the provided demo programs `nc16fc_minimal_example.c` (see Annex B), `nc16fc_compress.c` (see Annex C), `nc16fc_decompress.c` (see Annex D) and your own input data. No sample-data collection is included in the binary package. `nc16fc_minimal_example.c` is an example with full functionality but minimum complexity. `nc16fc_compress.c` and `nc16fc_decompress.c` illustrate an application where different compressors and image formats are supported when compressing images, and where the relevant decompressor is chosen based on information written into the file header. If the file extension is `tdms`, the `nc16fc_compress.c` assumes that the file is a TDMS file and it calls the function `nc16fc_copy_tdms_header`, which copies the header and metadata of the TDMS file, preventing data misalignment.

The libraries `nc16fc.a` (Linux, static) and `nc16fc.so`, `nc16fc.lib` (Windows, import library) and `nc16fc.dll` (Windows, shared) contain a variety of demo compressor implementations (see Annex A); calls are documented in the header file `nc16fc.h`.

Each of the compressor implementations is for a specific image format (bit depth, optional padding, color ordering) and a specific compression algorithm, but can handle variable image sizes `nx`, `ny`. The name of the compressor relates to the algorithm and to the image format supported, e.g. `TUNA_12BIT_CSI2BAYER` or `DOLPHIN_16BIT_BIGENDIAN`, see Annex A for more details.

For interleaved multi-channel input, compressor names use the suffixes `_2CHANNELS`, `_3CHANNELS`, or `_4CHANNELS`. For example, `MANTA_8BIT_2CHANNELS` processes samples consisting of two adjacent 8-bit channels. Likewise, `MANTA_16BIT` processes one sequence of 16-bit samples, while `MANTA_16BIT_2CHANNELS` processes two alternating 16-bit channels.

`MANTA`, `DOLPHIN` and `ORCA` support 8-bit input with one, two, three, or four channels, packed CSI-2 12-bit data, 16-bit little-endian data with one to four channels, 16-bit big-endian data with one or two channels, and UYVY input. `TUNA` supports packed CSI-2 12-bit data and 16-bit little- and big-endian data with one or two channels.

Signed samples have their own names with `_SIGNED`, e.g. `MANTA_16BIT_SIGNED` or `ORCA_8BIT_SIGNED_3CHANNELS`, for every 8-bit and 16-bit little-endian format of a compressor. For 16-bit data centred on zero (signed) or close to zero (unsigned), such as radar raw data, `DOLPHIN` has `DOLPHIN_16BIT_NEARZERO_SIGNED` and `DOLPHIN_16BIT_NEARZERO`, each also with `_2CHANNELS` to `_4CHANNELS`. Running `nc16fc_compress` without arguments lists the compressors a package contains.

7.1 Compressor / decompressor calls

Calls are defined in the header file `nc16fc.h` which needs to be included with:

```
#include "nc16fc.h"
```

A configuration is set up for one compressor and image size. Ask for the memory it needs, provide it, and set it up:

```
long long size = nc16fc_memory_size(DOLPHIN_12BIT_CSI2BAYER, nx, ny);
if (size < 0) {
    /* nc16fc_get_error_name(size): e.g. a format outside this package */
    return 1;
}
void *memory = malloc((size_t) size);      /* or a static array, or a pool */
nc16fc_Config config;
int status = nc16fc_init(&config, memory, size, DOLPHIN_12BIT_CSI2BAYER, nx,
ny);
if (status < 0) {
    /* nc16fc_get_error_name(status) */
    free(memory);
    return 1;
}
```

The compressors are constants of enum `Algorithm_Type` in `nc16fc.h`, e.g. `MANTA_16BIT`; `nc16fc_get_algorithm_type("MANTA_16BIT")` converts a name, for example one read from a file or a setting. Here `nx` is the number of pixels or interleaved multi-channel samples per row and `ny` is the image height. To change the compressor or the image size, give the configuration up with `nc16fc_deinit(config)` and set it up again, in the same memory if it is large enough. When a configuration is no longer needed:

```
nc16fc_deinit(config);
free(memory);
```

`nc16fc_init` returns 0, or a negative error and sets `config` to 0: `ERROR_INVALID_ARGUMENT` (no configuration or memory, or memory that overlaps a configuration in use), `ERROR_OUT_OF_MEMORY` (less memory than `nc16fc_memory_size` returned), `ERROR_CONFIGURATIONS_EXHAUSTED`, `ERROR_NO_PRIVILEGE_TO_SET_THIS_FORMAT` (a compressor outside this package), `ERROR_UNKNOWN_COMPRESSION_ALGORITHM` or `ERROR_BAD_IMAGE_DIMENSIONS`. `nc16fc_get_error_name(code)` returns the name of any error code; once a configuration exists, `nc16fc_get_error(config)` returns the last error of its calls.

ORCA codes images of up to 32768 pixels per row (65536 for 12-bit CSI-2 Bayer) and refuses wider images with `ERROR_BAD_IMAGE_DIMENSIONS`. The API accepts a positive image height up to `INT_MAX`, subject to size-overflow checks. The file tools restrict both dimensions to 1 through 65535. For explicit ORCA CSI-2 Bayer geometry, width and height must be even; for UYVY, width must be even. `ny = 1` selects automatic ORCA geometry as described below.

7.1.1 Automatic ORCA geometry detection

The compressor passed to `nc16fc_init` fixes the input format. Normally `nx` and `ny` specify the width and height. Setting `ny = 1` requests automatic geometry; pass a valid positive `nx`

placeholder, whose value is not used as the image width. Detection is performed for each compression call and its cost depends on the input and processor. Supply explicit dimensions when they are known.

Automatic decoding requires a stream created with geometry information. When an explicit encoder receives less than one complete frame, it copies those bytes without geometry information; decode such a raw-only stream with the same explicit dimensions. Raw tails after complete frames are supported in both decoding modes. The decoder handles streams produced by automatic mode without requiring the detected dimensions from the caller.

For TDMS files, the function `nc16fc_copy_tdms_header` (see above) should be called before starting the compression.

New configurations enable both the block header and checksum. Choose any different settings before calculating the output bound, and use matching settings for decoding; see Section 7.2.

A buffer with image frames is compressed via

```
csize = nc16fc_compress(config, image_buf, image_buf_size, compr_buf,
                        compr_buf_size)
```

where `image_buf` is a buffer containing one or multiple image frames, `image_buf_size` is the size in bytes of the data to be compressed, `compr_buf` is a buffer containing the compressed image(s) and `compr_buf_size` is the number of bytes allocated for `compr_buf`. The call processes the requested input bytes, including complete frames and any supported raw tail, and returns the compressed size in bytes or a negative error code.

For one complete frame, obtain `image_buf_size` from `nc16fc_get_frame_size(config)` so that all channels and the selected packing are included. For multiple frames, pass their total byte count. Obtain `compr_buf_size` from `nc16fc_compress_bound(config, image_buf_size)` and check for a negative error result before allocating `compr_buf`. Query the bound again after changing the input size or configuration. It covers calls with multiple frames as well. Incompressible input can produce more output than input. A buffer that turns out to be too small is reported as `ERROR_COMPRESSED_BUFFER_TOO_SMALL` and is never written past, so a smaller buffer may be used if the application handles that error.

A buffer with frames is decompressed via

```
dsize = nc16fc_decompress(config, image_buf, image_buf_size, compr_buf,
                          compr_buf_size)
```

Here, `compr_buf` is decompressed into `image_buf`. Pass the exact expected uncompressed size as `image_buf_size` and the available compressed size as `compr_buf_size`, and use the same `nc16fc_set_header` and `nc16fc_set_checksum` settings as when compressing. The return value is the number of compressed bytes consumed, or a negative error code. It is not the number of output bytes. Check errors and, for a complete compressed block, verify that all compressed bytes were consumed.

7.2 Block header and checksum

By default each non-empty block – the output of one `nc16fc_compress` call, which may contain multiple frames and a partial tail – starts with a header and checksum totalling 16 bytes. Zero-length calls return 0 and write no header or checksum. `nc16fc_compress_bound` includes the enabled prefix.

Bytes	Content	Purpose
0	block format version, 1	identifies the block layout
1–2	stable compressor/format ID, little endian	checks that the selected decoder matches
3–7	reserved, zero	
8–15	XXH3-64 (seed 0) of all uncompressed bytes, little endian	checks the decoded data

`nc16fc_set_header(config, enabled)` and `nc16fc_set_checksum(config, enabled)` control the two parts independently. Both accept only 0 (off) or 1 (on), return 0 on success or a negative error, and apply to compression and decompression. Invalid values leave the setting unchanged. Selecting another compressor preserves these settings; a newly acquired configuration starts with both on. With only one part enabled the prefix is 8 bytes; with both disabled it is absent.

Use the same settings when writing and reading. They are not detected automatically or recorded as flags in the block. For a legacy raw stream, switch both off on the encoder or decoder concerned:

```
int status = nc16fc_set_header(config, 0);
if (status == 0) status = nc16fc_set_checksum(config, 0);
if (status < 0) {
    /* Inspect nc16fc_get_error(config) and handle the error. */
    nc16fc_deinit(config);
    return 1;
}
```

Use 1 instead to enable either part again. Query `nc16fc_compress_bound` again after changing a setting. **Streams from versions before the block-header extension require both switches off.** The C and Python file examples provide `--legacy` for this purpose. A raw TUNA FPGA stream also requires both switches off; the underlying TUNA codec format is unchanged.

To write data for an older decoder, disable both parts on the encoder as well.

With the checksum enabled, pass the exact complete uncompressed size of the block to `nc16fc_decompress`. The header checks the compressor/format ID; it does not store the dimensions or uncompressed length, which the caller must provide correctly.

Error	Meaning
-------	---------

ERROR_END_OF_COMPRESSED_BUFFER_REACHED	not enough bytes for the enabled prefix or required payload
ERROR_STREAM_HEADER_INVALID	unknown block version or a nonzero reserved header byte
ERROR_STREAM_FORMAT_MISMATCH	the block ID differs from the selected compressor/format
ERROR_CHECKSUM_MISMATCH	the decoded bytes do not match the stored checksum

Other negative codec errors can occur for damaged payloads. Prefix/header rejection happens before writing decoded data. A checksum is checked after decoding, so on ERROR_CHECKSUM_MISMATCH the output buffer has already been written and must be discarded. Do not use output from a failed decode. Successful decoding returns the compressed bytes consumed, including the prefix; if the input is meant to contain exactly one block, verify that the return value equals its supplied compressed size.

After licence expiry, attempts that enter a codec count their requested output size towards the decoder's speed limit, including payload or checksum errors. Argument and block-header rejection happens before decoding and incurs no pacing delay. If the operating system cannot enforce the delay after decoding, the library clears the output buffer and returns ERROR_RATE_LIMIT_UNAVAILABLE.

XXH3 detects accidental data corruption; it is not cryptographic authentication. It does not protect external metadata, block ordering, or bytes outside the decoded block. Disabling the checksum removes this data-integrity check; disabling the header removes the compressor-ID check.

The example programs' outer **file header** is separate: it stores the algorithm name, dimensions and other file metadata. Their final incomplete frame and optional TDMS metadata are copied outside library compression calls and therefore carry no library checksum. The example file format does not provide a checksum for the entire file.

The checksum costs time in proportion to the uncompressed size, about 25–35 GB/s on the reference machine. For the fastest decoders this is noticeable: in the benchmark set, the decompression throughput of DOLPHIN and TUNA drops by about 21 %, of MANTA by 9 % and of ORCA by 3 %; compression throughput by 2–11 %. Where the application protects its data otherwise, the checksum can be switched off.

nc16fc_compress.c and nc16fc_decompress.c (see Annex C and D) are example programs that show how to use the library. The source nc16fc_benchmark.c contains a more extended program to benchmark compressors.

7.3 Compiling and linking a program in Linux using the library

Run Make from the extracted package directory:

```
make
./build/static/nc16fc_minimal_example
make CC=clang LINKAGE=shared
./build/shared/nc16fc_minimal_example
```

The default links the examples against `nc16fc.a`; `LINKAGE=shared` uses `nc16fc.so`. Both use the normal C runtime of the local build system. GCC and Clang are supported. `CC`, `CPPFLAGS`, `CFLAGS`, `LDFLAGS` and `LDLIBS` can be customized. Run `make clean` (or `make LINKAGE=shared clean`) before rebuilding a configuration with different compiler flags. Cleaning removes generated programs and runtime copies, preserving the supplied libraries and sources.

Static and shared outputs are separate in `build/static/` and `build/shared/`. For shared builds, Make copies `nc16fc.so` beside the programs and sets a relative runtime search path. For your own build system, the shared library has no `lib` prefix: use `-L. -l:nc16fc.so` rather than `-lnc16fc`. C++ integrations can include `nc16fc.h` directly. Link with the normal compiler driver so that the static library's compiler runtime helpers are resolved.

7.4 Compiling and linking a program in Windows using the library

The Windows DLL exposes a C interface and can be used from Visual Studio C or C++ projects. Select the **x64** platform, add the package directory to the include and library paths, and link against `nc16fc.lib`. Copy `nc16fc.dll` beside the executable. MinGW/MSYS2 is not required for this integration.

To build a supplied C example, open an **x64 Native Tools Command Prompt** for Visual Studio 2022 (C++ development tools and Windows SDK installed), change to the package directory, and run:

```
cl /nologo /O2 /TC /std:c11 /D_CRT_SECURE_NO_WARNINGS /I. ^
nc16fc_minimal_example.c nc16fc.lib ^
/Fe:nc16fc_minimal_example.exe
nc16fc_minimal_example.exe
```

The same command can build `nc16fc_compress.c`, `nc16fc_decompress.c` or `nc16fc_benchmark.c` with the corresponding executable name. The GNU Makefile requires GNU Make; Visual Studio's `nmake` is not a replacement for it.

Alternatively, in an MSYS2 shell with MinGW-w64 and Make installed, run:

```
make CC=gcc
./build/shared/nc16fc_minimal_example.exe
```

The Makefile links against `nc16fc.lib` and places programs and a copy of `nc16fc.dll` in `build/shared/`. Keep that DLL beside the programs. The import library is not a standalone static codec library. No Linux syscall report is required for a Windows package.

7.5 Running the supplied programs in Linux

Build the examples first, then run the commands from the extracted package directory. Supply your own packed 12-bit CSI-2 Bayer input with 1920 by 1088 pixels per frame; `img_1920x1088_12bit` below is a placeholder filename, not a supplied dataset:

```
./build/static/nc16fc_compress DOLPHIN_12BIT_CSI2BAYER 1920 1088
img_1920x1088_12bit a
./build/static/nc16fc_decompress a b
```

`a` is the compressed file, and `b` is the decompressed file. One can check if the decompressed file is equal to the original file:

```
cmp img_1920x1088_12bit b
```

The library prints nothing; the programs show its licence terms and the compressors of the package when asked:

```
./build/static/nc16fc_compress --license
./build/static/nc16fc_compress --algorithms
```

7.6 Running the supplied programs in Windows

Build the examples first, then run the commands from the extracted package directory. Supply your own packed 12-bit CSI-2 Bayer input with 1920 by 1088 pixels per frame; `img_1920x1088_12bit` below is a placeholder filename, not a supplied dataset:

```
build\shared\nc16fc_compress.exe DOLPHIN_12BIT_CSI2BAYER 1920 1088
img_1920x1088_12bit a
build\shared\nc16fc_decompress.exe a b
```

`a` is the compressed file, and `b` is the decompressed file. Use the Windows command prompt (`cmd.exe`) to compare the files:

```
fc /b img_1920x1088_12bit b
```

The benchmark tool measures the configured combinations of algorithm, geometry and dataset. It does not automatically test all formats in the library. Run:

```
./build/static/nc16fc_benchmark --help
./build/static/nc16fc_benchmark --list
./build/static/nc16fc_benchmark 0.1
```

The final command reads the listed input files below `data/`, relative to the current directory. They come in the separate data archive `nc16fc_data_<name>_<version>.tar.gz`, when it is supplied: extract it in the package directory (`tar xzf nc16fc_data_...tar.gz` creates `data/`, with `data/README.txt` describing the layout of every file). Each file must be

nonempty and contain a whole number of frames of its listed format. For your own data, adjust the dataset entries in `nc16fc_benchmark.c` and rebuild it. The argument is a positive minimum duration in seconds per compression/decompression measurement (default 3); timings are checked in batches of 100 operations. Speeds are reported in MiB/s, using uncompressed bytes, and ratio means compressed size divided by input size.

7.7 Accessing the library from Python

The Python scripts `nc16fc_compress.py` and `nc16fc_decompress.py` use the same file format and positional arguments as the C programs. The `--license` and `--algorithms` options shown above belong to the C tools; `--legacy` (blocks without header and checksum, for data of earlier versions) works in both. Keep `nc16fc.py` and the shared library beside the scripts. The scripts load `nc16fc.so` on Linux and `nc16fc.dll` on Windows from their own directory.

```
python3 nc16fc_compress.py DOLPHIN_12BIT_CSI2BAYER 1920 1088 img_1920x1088_12bit
a
python3 nc16fc_decompress.py a b
```

The compressors produce exactly the same output:

```
./build/static/nc16fc_compress DOLPHIN_12BIT_CSI2BAYER 1920 1088
img_1920x1088_12bit a
python3 nc16fc_decompress.py a b
cmp img_1920x1088_12bit b
```

8 Your data always stays readable

A time-limited library runs at full speed until its licence ends; an unlimited package has no expiry. The applicable expiry date is available from the library's licence notice (`nc16fc_get_license()`), rather than this manual. The following expiry behaviour applies to time-limited packages. The trial starts when the package is built, not on its first use. After that date it still **decodes compatible streams for the formats included in this package**: it switches by itself to a slower decoder that is part of every time-limited library. You never lose access to your data because a licence has ended.

Compression ends at that date: `nc16fc_compress` returns `ERROR_TEST_VERSION_EXPIRED`. There is no compression mode after expiry. The library never ends the application and prints nothing; `nc16fc_memory_size`, `nc16fc_init` and `nc16fc_deinit` work as before.

A system clock that is set before the package date – for example a device that starts at 1970 and sets its clock later by NTP or GPS – or that cannot be read does not end the licence. While the clock is in that state, `nc16fc_compress` returns `ERROR_LICENSE_CLOCK_INVALID` and decompression uses the slower decoder. Once the clock is set, compression works again without a restart.

A time-limited shared library checks its own code. If it has been modified, `nc16fc_compress` returns `ERROR_LIBRARY_MODIFIED` and decompression uses the slower decoder.

The slower decoder has a configured maximum output rate shared by all decoder threads and configurations of the loaded library. The licence notice reports the limit in uncompressed bytes per second. The decoder sleeps as needed; the limit is not multiplied by creating more configurations or threads. Decoding time counts towards the wait. Before expiry, decoding runs at full speed.

`compiled.slow_decoder_bytes_per_second` in `PACKAGE.json` records the shared decoder output limit after expiry. It does not enable compression after expiry.

9 Licence notice

The library prints nothing. Its notice – version, copyright and licence terms and the expiry date of a demo version – and the compressors of the package are texts the application can ask for:

```
printf("%s", nc16fc_get_license());  
printf("%s\n", nc16fc_get_algorithms());
```

10 Support contact

E-Mail: support@nanocode.com

Phone: +49 6226/7876840

11 Annex A: Included formats and representative benchmarks

The following formats are enabled in this package. This list is generated from the same configuration as the library; `nc16fc_compress --algorithms` queries it at runtime. The benchmark table further below illustrates representative product measurements; those datasets are not included in this binary package.

DOLPHIN_12BIT_CSI2BAYER	MANTA_8BIT_4CHANNELS
DOLPHIN_16BIT	MANTA_12BIT_CSI2BAYER
DOLPHIN_16BIT_BIGENDIAN	MANTA_16BIT
DOLPHIN_16BIT_2CHANNELS	MANTA_16BIT_2CHANNELS
DOLPHIN_16BIT_BIGENDIAN_2CHANNELS	MANTA_16BIT_BIGENDIAN
DOLPHIN_8BIT	MANTA_16BIT_BIGENDIAN_2CHANNELS
DOLPHIN_8BIT_2CHANNELS	MANTA_16BIT_YUV422_UYVY
DOLPHIN_8BIT_3CHANNELS	MANTA_16BIT_3CHANNELS
DOLPHIN_8BIT_4CHANNELS	MANTA_16BIT_4CHANNELS
DOLPHIN_16BIT_YUV422_UYVY	MANTA_8BIT_SIGNED
DOLPHIN_16BIT_3CHANNELS	MANTA_8BIT_SIGNED_2CHANNELS
DOLPHIN_16BIT_4CHANNELS	MANTA_8BIT_SIGNED_3CHANNELS
DOLPHIN_8BIT_SIGNED	MANTA_8BIT_SIGNED_4CHANNELS
DOLPHIN_8BIT_SIGNED_2CHANNELS	MANTA_16BIT_SIGNED
DOLPHIN_8BIT_SIGNED_3CHANNELS	MANTA_16BIT_SIGNED_2CHANNELS
DOLPHIN_8BIT_SIGNED_4CHANNELS	MANTA_16BIT_SIGNED_3CHANNELS
DOLPHIN_16BIT_SIGNED	MANTA_16BIT_SIGNED_4CHANNELS
DOLPHIN_16BIT_SIGNED_2CHANNELS	ORCA_8BIT
DOLPHIN_16BIT_SIGNED_3CHANNELS	ORCA_8BIT_2CHANNELS
DOLPHIN_16BIT_SIGNED_4CHANNELS	ORCA_8BIT_3CHANNELS
DOLPHIN_16BIT_NEARZERO	ORCA_8BIT_4CHANNELS
DOLPHIN_16BIT_NEARZERO_2CHANNELS	ORCA_12BIT_CSI2BAYER
DOLPHIN_16BIT_NEARZERO_3CHANNELS	ORCA_16BIT
DOLPHIN_16BIT_NEARZERO_4CHANNELS	ORCA_16BIT_2CHANNELS
DOLPHIN_16BIT_NEARZERO_SIGNED	ORCA_16BIT_3CHANNELS
DOLPHIN_16BIT_NEARZERO_SIGNED_2CHANNELS	ORCA_16BIT_4CHANNELS
DOLPHIN_16BIT_NEARZERO_SIGNED_3CHANNELS	ORCA_16BIT_BIGENDIAN
DOLPHIN_16BIT_NEARZERO_SIGNED_4CHANNELS	ORCA_16BIT_YUV422_UYVY
TUNA_12BIT_CSI2BAYER	ORCA_16BIT_BIGENDIAN_2CHANNELS
TUNA_16BIT	ORCA_8BIT_SIGNED
TUNA_16BIT_BIGENDIAN	ORCA_8BIT_SIGNED_2CHANNELS
TUNA_16BIT_2CHANNELS	ORCA_8BIT_SIGNED_3CHANNELS
TUNA_16BIT_BIGENDIAN_2CHANNELS	ORCA_8BIT_SIGNED_4CHANNELS
TUNA_16BIT_SIGNED	ORCA_16BIT_SIGNED
TUNA_16BIT_SIGNED_2CHANNELS	ORCA_16BIT_SIGNED_2CHANNELS
MANTA_8BIT	ORCA_16BIT_SIGNED_3CHANNELS
MANTA_8BIT_2CHANNELS	ORCA_16BIT_SIGNED_4CHANNELS
MANTA_8BIT_3CHANNELS	

MANTA balances a good compression ratio with high speed and is a suitable general-purpose default. DOLPHIN prioritizes throughput, low latency and a small memory footprint. TUNA offers an FPGA IP core for compatible compression and decompression on devices and hosts. ORCA targets a high compression ratio for image data, with greater memory requirements and lower throughput.

DOLPHIN's `_NEARZERO` formats are intended for 16-bit data centred on or close to zero, such as radar samples. `_SIGNED` formats accept signed samples. Choose the format that

matches the input representation; compression is lossless and preserves the input bytes. Accelerated and scalar implementations produce compatible streams.

Support for additional input formats can be added upon request.

The following table contains representative product benchmarks, one thread each, on an AMD EPYC 9645. Speeds depend on the data and the processor. The radar file (signed 16-bit samples) is coded by the 16BIT_SIGNED format of every compressor and by DOLPHIN's near-zero format. It is 1 MiB and stays in the processor cache, so its speeds are upper bounds.

Algorithm	Data file	Ratio	v_compr [MB/s]	v_decompr [MB/s]
MANTA_8BIT	frame_01_8Mpx_3840x2160_y.yuv	0.612	1174	2018
MANTA_8BIT_2CHANNELS	frame_01_8Mpx_3840x2160_uv.yuv	0.175	1013	2065
MANTA_12BIT_CSI2BAYER	img_1920x1088_12bit	0.534	980	1488
MANTA_16BIT_2CHANNELS	02_16bit_LE_2channels_816x1288.raw	0.670	1187	2278
MANTA_16BIT_BIGENDIAN	03_12bit_BE_1channel_1280x1014.raw	0.554	1264	2390
MANTA_16BIT_YUV422_UYVY	frame_01_8Mpx_3840x2160.yuv	0.517	1462	3103
MANTA_16BIT_3CHANNELS	04_12bit_LE_3channels_816x644.raw	0.326	1798	2872
MANTA_16BIT_4CHANNELS	05_12bit_LE_4channels_816x644.raw	0.323	1552	2687
MANTA_16BIT_BIGENDIAN_2CHANNELS	06_16bit_BE_2channels_816x1288.raw	0.670	1243	2501
MANTA_16BIT_SIGNED	07_radar_int16_LE_512x1024.raw	0.530	1575	2486
DOLPHIN_8BIT	frame_01_8Mpx_3840x2160_y.yuv	0.691	2146	2577
DOLPHIN_8BIT_2CHANNELS	frame_01_8Mpx_3840x2160_uv.yuv	0.337	2367	4330
DOLPHIN_12BIT_CSI2BAYER	img_1920x1088_12bit	0.558	2064	4822
DOLPHIN_16BIT_2CHANNELS	02_16bit_LE_2channels_816x1288.raw	0.720	2799	5234
DOLPHIN_16BIT_BIGENDIAN	03_12bit_BE_1channel_1280x1014.raw	0.587	4167	5282
DOLPHIN_16BIT_YUV422_UYVY	frame_01_8Mpx_3840x2160.yuv	0.528	2803	4100
DOLPHIN_16BIT_3CHANNELS	04_12bit_LE_3channels_816x644.raw	0.438	5412	8755
DOLPHIN_16BIT_4CHANNELS	05_12bit_LE_4channels_816x644.raw	0.438	5077	12386
DOLPHIN_16BIT_BIGENDIAN_2CHANNELS	06_16bit_BE_2channels_816x1288.raw	0.720	2884	5083
DOLPHIN_16BIT_SIGNED	07_radar_int16_LE_512x1024.raw	0.563	11175	17775
DOLPHIN_16BIT_NEARZERO_SIGNED	07_radar_int16_LE_512x1024.raw	0.515	28270	42133
TUNA_12BIT_CSI2BAYER	img_1920x1088_12bit	0.556	2028	6187
TUNA_16BIT_2CHANNELS	02_16bit_LE_2channels_816x1288.raw	0.689	2571	9100
TUNA_16BIT_BIGENDIAN	03_12bit_BE_1channel_1280x1014.raw	0.562	2866	8074
TUNA_16BIT_BIGENDIAN_2CHANNELS	06_16bit_BE_2channels_816x1288.raw	0.689	2652	9940
TUNA_16BIT_SIGNED	07_radar_int16_LE_512x1024.raw	0.526	3457	9283
ORCA_8BIT	frame_01_8Mpx_3840x2160_y.yuv	0.524	328	399
ORCA_8BIT_2CHANNELS	frame_01_8Mpx_3840x2160_uv.yuv	0.130	266	300
ORCA_12BIT_CSI2BAYER	img_1920x1088_12bit	0.443	312	380
ORCA_16BIT_2CHANNELS	02_16bit_LE_2channels_816x1288.raw	0.610	339	421
ORCA_16BIT_BIGENDIAN	03_12bit_BE_1channel_1280x1014.raw	0.526	372	430
ORCA_16BIT_YUV422_UYVY	frame_01_8Mpx_3840x2160.yuv	0.327	285	315
ORCA_16BIT_3CHANNELS	04_12bit_LE_3channels_816x644.raw	0.329	512	627
ORCA_16BIT_4CHANNELS	05_12bit_LE_4channels_816x644.raw	0.326	474	617
ORCA_16BIT_BIGENDIAN_2CHANNELS	06_16bit_BE_2channels_816x1288.raw	0.610	379	449
ORCA_16BIT_SIGNED	07_radar_int16_LE_512x1024.raw	0.551	390	537

12 Annex B: Minimal example program

The example program `nc16fc_minimal_example.c` illustrates initialization, buffer allocation, compression, decompression, error handling and a byte-for-byte roundtrip check. Run `./build/static/nc16fc_minimal_example` without any data files: it creates a synthetic 1632 by 1288 single-channel 16-bit little-endian frame in memory. Alternatively, pass a filename to read the first complete frame of those dimensions. Synthetic results demonstrate API use and are not representative performance measurements.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "nc16fc.h"

int main(int argc, char **argv)
{
    nc16fc_Config config;
    FILE *f;
    void *memory;
    unsigned char *data, *data2, *compr;
    long long memory_size, data_size, csize, csize2, compr_size;
    const long long nx = 1632; // image width
    const long long ny = 1288; // image height
    int error;

    if (argc > 2) nc16fc_die("SYNTAX: %s [16bit-little-endian-input.raw]", argv[0]);

    // the memory for one configuration of this format and size; the library allocates nothing
    memory_size = nc16fc_memory_size(DOLPHIN_16BIT, nx, ny);
    if (memory_size < 0) nc16fc_die("%s", nc16fc_get_error_name(memory_size));
    memory = malloc((size_t) memory_size);
    if (!memory) nc16fc_die("out of memory");
    error = nc16fc_init(&config, memory, memory_size, DOLPHIN_16BIT, nx, ny);
    if (error < 0) nc16fc_die("%s", nc16fc_get_error_name(error));

    data_size = nc16fc_get_frame_size(config); // includes channels and packing
    compr_size = nc16fc_compress_bound(config, data_size);
    if (data_size < 0 || compr_size < 0) nc16fc_die("%s", nc16fc_get_error(config));
    data = (unsigned char *) malloc((size_t) data_size);
    data2 = (unsigned char *) malloc((size_t) data_size);
    compr = (unsigned char *) malloc((size_t) compr_size);
    if (!data || !compr || !data2) nc16fc_die("out of memory");

    if (argc == 2) {
        f = fopen(argv[1], "rb");
        if (!f) nc16fc_die("cannot open %s", argv[1]);
        if (fread(data, 1, (size_t) data_size, f) != (size_t) data_size)
            nc16fc_die("cannot read one complete frame from %s", argv[1]);
        fclose(f);
    } else {
        // Self-contained API demonstration, not a representative speed benchmark.
        for (long long i = 0; i < data_size / 2; ++i) {
            unsigned value = (unsigned) (i % nx) & 4095;
            data[2*i] = (unsigned char) value;
            data[2*i+1] = (unsigned char) (value >> 8);
        }
    }

    csize = nc16fc_compress(config, data, data_size, compr, compr_size);
    if (csize < 0) nc16fc_die("%s", nc16fc_get_error(config));
    csize2 = nc16fc_decompress(config, data2, data_size, compr, csize);
    if (csize2 < 0) nc16fc_die("%s", nc16fc_get_error(config));
    if (csize2 != csize || memcmp(data, data2, (size_t) data_size))
        nc16fc_die("roundtrip mismatch");
    nc16fc_deinit(config); // give the configuration up before freeing its memory
    free(memory);

    printf("roundtrip OK; compression ratio: %.3f\n", (double) csize / (double) data_size);
    free(data);
    free(data2);
    free(compr);
}
```

```
    return 0;  
}
```

13 Annex C: Example compressor

The example program `nc16fc_compress.c` shows how to specify the compression algorithm on the command line and store it in the file header.

The file header records the library version, algorithm name, dimensions and TDMS flag. It is a wrapper used by the example tools, separate from the per-call block header and checksum returned by `nc16fc_compress` (see Section 7.2). The decompressor selects the format from this header. CPU dispatch is automatic: there is no separate customer-selectable “fast” algorithm name. The scalar and accelerated kernels use the same codec format.

```
#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#ifdef _WIN32
#include <io.h>
#include <windows.h>
#else
#include <unistd.h>
#endif

#include "nc16fc.h"

/* Open without truncating, then check file identity before replacing output. */
static FILE *nc16fc_open_output(FILE *input, const char *path)
{
    FILE *output;
#ifdef _WIN32
    int fd = _open(path, _O_WRONLY | _O_CREAT | _O_BINARY, _S_IREAD | _S_IWRITE);
    if (fd < 0) nc16fc_die("ERROR: could not open '%s': %s", path, strerror(errno));
    HANDLE ih = (HANDLE) _get_osfhandle(_fileno(input));
    HANDLE oh = (HANDLE) _get_osfhandle(fd);
    int regular = GetFileType(oh) == FILE_TYPE_DISK;
    if (regular && GetFileType(ih) == FILE_TYPE_DISK) {
        BY_HANDLE_FILE_INFORMATION a, b;
        if (!GetFileInformationByHandle(ih, &a) || !GetFileInformationByHandle(oh, &b)) {
            _close(fd);
            nc16fc_die("ERROR: could not check input/output file identity");
        }
        if (a.dwVolumeSerialNumber == b.dwVolumeSerialNumber &&
            a.nFileIndexHigh == b.nFileIndexHigh && a.nFileIndexLow == b.nFileIndexLow) {
            _close(fd);
            nc16fc_die("ERROR: input and output name the same file");
        }
    }
    output = _fdopen(fd, "wb");
    if (!output) { _close(fd); nc16fc_die("ERROR: could not open output stream '%s'", path); }
    if (regular && _chsize_s(fd, 0) != 0)
        nc16fc_die("ERROR: could not truncate '%s'", path);
#else
    int fd = open(path, O_WRONLY | O_CREAT, 0666);
    struct stat a, b;
    if (fd < 0) nc16fc_die("ERROR: could not open '%s': %s", path, strerror(errno));
    if (fstat(fileno(input), &a) != 0 || fstat(fd, &b) != 0) {
        close(fd);
        nc16fc_die("ERROR: could not check input/output file identity");
    }
    if (a.st_dev == b.st_dev && a.st_ino == b.st_ino) {
        close(fd);
        nc16fc_die("ERROR: input and output name the same file");
    }
    output = fdopen(fd, "wb");
    if (!output) { close(fd); nc16fc_die("ERROR: could not open output stream '%s'", path); }
    /* Devices such as /dev/full and pipes are streams, not files to truncate. */
    if (S_ISREG(b.st_mode) && ftruncate(fd, 0) != 0)
        nc16fc_die("ERROR: could not truncate '%s': %s", path, strerror(errno));
#endif
}
```

```

    return output;
}

/* the formats of this package, which the library tells when asked */
static void print_algorithms(void)
{
    fprintf(stderr, "Available algorithms: %s\n\n", ncl6fc_get_algorithms());
}

int main(int argc, char **argv)
{
    ncl6fc_Config config;
    void *memory;
    unsigned char *img, *compr;
    unsigned int version;
    long long nbytes, dn, cn, nx, ny, csize = 0, imgsize, compr_size, memory_size;
    char compressor[128], *file_in, *file_out, *ext, tdms;
    int format, error;
    FILE *fi, *fo;

    /* the library prints nothing; its notice and its formats are asked for */
    if (argc == 2 && !strcmp(argv[1], "--license")) { printf("%s", ncl6fc_get_license()); return 0; }
    if (argc == 2 && !strcmp(argv[1], "--algorithms")) { printf("%s\n", ncl6fc_get_algorithms()); return 0; }

    /* --legacy: blocks without header and checksum, readable by earlier ncl6fc */
    int legacy = argc > 1 && !strcmp(argv[1], "--legacy");
    if (legacy) { argv++; argc--; }

    if (argc != 6) {
        fprintf(stderr, "ERROR: wrong number of arguments\n");
        fprintf(stderr, "SYNTAX: ncl6fc_compress [--legacy] COMPRESSOR IMAGE_WIDTH IMAGE_HEIGHT INPUT_FILE OUTPUT_FILE\n");
        fprintf(stderr, "    --legacy: blocks without header and checksum, for earlier ncl6fc versions\n");
        fprintf(stderr, "    COMPRESSOR: which compressor (e.g. MANTA_8BIT_2CHANNELS or MANTA_12BIT_CSI2BAYER)\n");
        fprintf(stderr, "    IMAGE_WIDTH: number of pixels or interleaved samples per row\n");
        fprintf(stderr, "    IMAGE_HEIGHT: height of image in pixels\n");
        fprintf(stderr, "    INPUT_FILE: filename of input image\n");
        fprintf(stderr, "    OUTPUT_FILE: filename of output image\n");
        fprintf(stderr, "ncl6fc_compress --license shows the licence terms, --algorithms the formats.\n");
        print_algorithms();
        exit(1);
    }

    strncpy(compressor, argv[1], sizeof(compressor) - 1);
    compressor[sizeof(compressor) - 1] = '\0';
    nx = atoll(argv[2]);
    ny = atoll(argv[3]);
    if (nx <= 0 || ny <= 0 || nx > 65535 || ny > 65535) ncl6fc_die("ERROR: invalid image dimensions %lldx%lld", nx, ny); //
    ncl6fc_decompress rejects the same

    /* the configuration: its memory comes from here, the library allocates nothing */
    format = ncl6fc_get_algorithm_type(compressor);
    memory_size = format < 0 ? format : ncl6fc_memory_size(format, nx, ny);
    if (memory_size < 0) {
        print_algorithms();
        ncl6fc_die("%s", ncl6fc_get_error_name(memory_size));
    }
    memory = malloc((size_t) memory_size);
    if (!memory) ncl6fc_die("ERROR: out of memory");
    error = ncl6fc_init(&config, memory, memory_size, format, nx, ny);
    if (error < 0) ncl6fc_die("%s", ncl6fc_get_error_name(error));
    if (legacy && (ncl6fc_set_header(config, 0) < 0 || ncl6fc_set_checksum(config, 0) < 0))
        ncl6fc_die("%s", ncl6fc_get_error(config));

    file_in = argv[4];
    fi = fopen(file_in, "rb");
    if (!fi) ncl6fc_die("could not open '%s' for reading", file_in);
    ext = strrchr(file_in, '.');
    if (ext == 0) tdms = 0; else tdms = !strcmp(ext+1, "tdms");
    if (tdms == 1) fprintf(stderr, "NOTE: TDMS file detected.\n");

    file_out = argv[5];
    fo = ncl6fc_open_output(fi, file_out);

    imgsize = ncl6fc_get_frame_size(config);
    if (imgsize <= 0) ncl6fc_die("ERROR: invalid image size: %s", ncl6fc_get_error(config));
    compr_size = ncl6fc_compress_bound(config, imgsize);
    if (compr_size < 0) ncl6fc_die("%s", ncl6fc_get_error(config));
    img = (unsigned char *) malloc((size_t) imgsize);
    compr = (unsigned char *) malloc((size_t) compr_size);

```

```

if (img == NULL || compr == NULL) ncl6fc_die("ERROR: out of memory");

version = ncl6fc_get_version(config);

if (fwrite(&version, 4, 1, fo) != 1) ncl6fc_die("ERROR: could not write version");
if (fwrite(compressor, 128, 1, fo) != 1) ncl6fc_die("ERROR: could not write name of algorithm"); // could also store
integer ID
if (fwrite(&nx, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write image width");
if (fwrite(&ny, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write image height");
if (fwrite(&tdms, 1, 1, fo) != 1) ncl6fc_die("ERROR: could not write TDMS flag");

if (tdms == 1) if (ncl6fc_copy_tdms_header(config, fi, fo) != 0) ncl6fc_die("ERROR: could not copy TDMS header: %s",
ncl6fc_get_error(config));

double t0 = ncl6fc_tic();
int benchmark_loops = 1;
dn = cn = 0;
while (1) {
    nbytes = (long long) fread(img, 1, (size_t) imgsize, fi); // read image
    if (ferror(fi)) ncl6fc_die("ERROR: could not read '%s': %s", file_in, strerror(errno));
    if (nbytes != imgsize) { // partial image or end of stream
        long long x = -1;
        if (fwrite(&x, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write imgsize (-1)");
        if (fwrite(&nbytes, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write csize (size of partial image)");
        if (fwrite(img, 1, (size_t) nbytes, fo) != (size_t) nbytes) ncl6fc_die("ERROR: could not write the partial
image");
        break;
    }
    for (int y = 0; y < benchmark_loops; y++)
        csize = ncl6fc_compress(config, img, imgsize, compr, compr_size);
    if (csize < 0) ncl6fc_die("%s", ncl6fc_get_error(config));
    if (fwrite(&nbytes, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write size of img block");
    if (fwrite(&csize, 8, 1, fo) != 1) ncl6fc_die("ERROR: could not write size of compressed block");
    if (fwrite(compr, 1, (size_t) csize, fo) != (size_t) csize) ncl6fc_die("ERROR: could not write the compressed
buffer");
    dn += nbytes;
    cn += csize;
    fprintf(stderr, "uncompressed: %10lld    compressed: %10lld    ratio: %.4f\n", nbytes, csize, (double)csize/
(double)nbytes);
}
fclose(fi);
if (fclose(fo) != 0) ncl6fc_die("ERROR: could not finish writing '%s': %s", file_out, strerror(errno));
double dt = ncl6fc_toc(t0);
fprintf(stderr, "SUMMARY: uncompressed: %10lld    compressed: %10lld    ratio: %.4f    speed: %.0f MiB/s\n", dn, cn, dn >
0 ? (double)cn/(double)dn : 0.0, dt > 0 ? (double)benchmark_loops*(double)dn/dt/1048576.0 : 0.0);
free(compr);
free(img);
ncl6fc_deinit(config);
free(memory);
return 0;
}

```

14 Annex D: Example decompressor

The example program `nc16fc_decompress.c` shows how to read a file compressed with `nc16fc_compress.c` and select the correct decompression algorithm during runtime.

```
#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#ifdef _WIN32
#include <io.h>
#include <windows.h>
#else
#include <unistd.h>
#endif

#include "nc16fc.h"

/* Open without truncating, then check file identity before replacing output. */
static FILE *nc16fc_open_output(FILE *input, const char *path)
{
    FILE *output;
#ifdef _WIN32
    int fd = _open(path, _O_WRONLY | _O_CREAT | _O_BINARY, _S_IREAD | _S_IWRITE);
    if (fd < 0) nc16fc_die("ERROR: could not open '%s': %s", path, strerror(errno));
    HANDLE ih = (HANDLE) _get_osfhandle(_fileno(input));
    HANDLE oh = (HANDLE) _get_osfhandle(fd);
    int regular = GetFileType(oh) == FILE_TYPE_DISK;
    if (regular && GetFileType(ih) == FILE_TYPE_DISK) {
        BY_HANDLE_FILE_INFORMATION a, b;
        if (!GetFileInformationByHandle(ih, &a) || !GetFileInformationByHandle(oh, &b)) {
            _close(fd);
            nc16fc_die("ERROR: could not check input/output file identity");
        }
        if (a.dwVolumeSerialNumber == b.dwVolumeSerialNumber &&
            a.nFileIndexHigh == b.nFileIndexHigh && a.nFileIndexLow == b.nFileIndexLow) {
            _close(fd);
            nc16fc_die("ERROR: input and output name the same file");
        }
    }
    output = _fdopen(fd, "wb");
    if (!output) { _close(fd); nc16fc_die("ERROR: could not open output stream '%s'", path); }
    if (regular && _chsize_s(fd, 0) != 0)
        nc16fc_die("ERROR: could not truncate '%s'", path);
#else
    int fd = open(path, O_WRONLY | O_CREAT, 0666);
    struct stat a, b;
    if (fd < 0) nc16fc_die("ERROR: could not open '%s': %s", path, strerror(errno));
    if (fstat(fileno(input), &a) != 0 || fstat(fd, &b) != 0) {
        close(fd);
        nc16fc_die("ERROR: could not check input/output file identity");
    }
    if (a.st_dev == b.st_dev && a.st_ino == b.st_ino) {
        close(fd);
        nc16fc_die("ERROR: input and output name the same file");
    }
    output = fdopen(fd, "wb");
    if (!output) { close(fd); nc16fc_die("ERROR: could not open output stream '%s'", path); }
    /* Devices such as /dev/full and pipes are streams, not files to truncate. */
    if (S_ISREG(b.st_mode) && ftruncate(fd, 0) != 0)
        nc16fc_die("ERROR: could not truncate '%s': %s", path, strerror(errno));
#endif
    return output;
}

int main(int argc, char **argv)
{
    nc16fc_Config config;
    void *memory;
    unsigned char *img, *compr;
    unsigned int version;
    long long nbytes = 0, dn, cn, nx, ny, csize, imgsize, compr_size, memory_size;
    char compressor[128], *file_in, *file_out, *tdms;
```

```

int format, error;
FILE *fi, *fo;

/* the library prints nothing; its notice and its formats are asked for */
if (argc == 2 && !strcmp(argv[1], "--license")) { printf("%s", ncl6fc_get_license()); return 0; }
if (argc == 2 && !strcmp(argv[1], "--algorithms")) { printf("%s\n", ncl6fc_get_algorithms()); return 0; }

/* --legacy: a file written by an earlier ncl6fc, blocks without header and checksum */
int legacy = argc > 1 && !strcmp(argv[1], "--legacy");
if (legacy) { argv++; argc--; }

if (argc != 3) {
    fprintf(stderr, "ERROR: wrong number of arguments\n");
    fprintf(stderr, "SYNTAX: ncl6fc_decompress [--legacy] INPUT_FILE OUTPUT_FILE\n");
    fprintf(stderr, "      --legacy: a file written by an earlier ncl6fc (no block header or checksum)\n");
    fprintf(stderr, "      INPUT_FILE: filename of input image\n");
    fprintf(stderr, "      OUTPUT_FILE: filename of output image\n");
    fprintf(stderr, "ncl6fc_decompress --license shows the licence terms, --algorithms the formats.\n");
    exit(1);
}

file_in = argv[1];
fi = fopen(file_in, "rb");
if (!fi) ncl6fc_die("could not open '%s' for reading", file_in);

file_out = argv[2];
fo = ncl6fc_open_output(fi, file_out);

if (fread(&version, 4, 1, fi) != 1) ncl6fc_die("ERROR: could not read version");
if (fread(compressor, 128, 1, fi) != 1) ncl6fc_die("ERROR: could not read name of algorithm");
compressor[sizeof(compressor) - 1] = '\0';
if (fread(&nx, 8, 1, fi) != 1) ncl6fc_die("ERROR: could not read image width");
if (fread(&ny, 8, 1, fi) != 1) ncl6fc_die("ERROR: could not read image height");
if (nx <= 0 || ny <= 0 || nx > 65535 || ny > 65535) ncl6fc_die("ERROR: invalid image dimensions %lldx%lld", nx, ny);
if (fread(&tdms, 1, 1, fi) != 1) ncl6fc_die("ERROR: could not read TDMS flag");

/* the configuration for the format named in the file; its memory comes from here */
format = ncl6fc_get_algorithm_type(compressor);
memory_size = format < 0 ? format : ncl6fc_memory_size(format, nx, ny);
if (memory_size < 0) {
    fprintf(stderr, "Available algorithms: %s\n\n", ncl6fc_get_algorithms());
    ncl6fc_die("%s", ncl6fc_get_error_name(memory_size));
}
memory = malloc((size_t) memory_size);
if (!memory) ncl6fc_die("ERROR: out of memory");
error = ncl6fc_init(&config, memory, memory_size, format, nx, ny);
if (error < 0) ncl6fc_die("%s", ncl6fc_get_error_name(error));
if (legacy && (ncl6fc_set_header(config, 0) < 0 || ncl6fc_set_checksum(config, 0) < 0))
    ncl6fc_die("%s", ncl6fc_get_error(config));

if (tdms == 1) if (ncl6fc_copy_tdms_header(config, fi, fo) != 0) ncl6fc_die("ERROR: could not copy TDMS header");

long long imgsize_full = ncl6fc_get_frame_size(config);
if (imgsize_full <= 0 || imgsize_full > (1LL<<40)) ncl6fc_die("ERROR: imgsize_full out of range: %lld", imgsize_full);
compr_size = ncl6fc_compress_bound(config, imgsize_full);
if (compr_size < 0) ncl6fc_die("%s", ncl6fc_get_error(config));
img = (unsigned char *) malloc((size_t) imgsize_full);
compr = (unsigned char *) malloc((size_t) compr_size);
if (img == NULL || compr == NULL) ncl6fc_die("ERROR: out of memory");

double t0 = ncl6fc_tic();
int benchmark_loops = 1;
dn = cn = 0;
while (1) {
    if (fread(&imgsize, 8, 1, fi) != 1) ncl6fc_die("ERROR: stream ended while reading imgsize");
    if (fread(&csize, 8, 1, fi) != 1) ncl6fc_die("ERROR: stream ended while reading csize");
    if (imgsize == -1) {
        if (csize < 0 || csize > imgsize_full) ncl6fc_die("ERROR: partial image size %lld out of bounds", csize);
        if (fread(img, 1, (size_t) csize, fi) != (size_t) csize) ncl6fc_die("ERROR: could not read the partial image");
        if (fwrite(img, 1, (size_t) csize, fo) != (size_t) csize) ncl6fc_die("ERROR: could not write the partial image");
        break;
    }
    if (imgsize <= 0 || imgsize > imgsize_full) ncl6fc_die("ERROR: imgsize %lld out of bounds (max %lld)", imgsize,
imgsize_full);
    if (csize <= 0 || csize > compr_size) ncl6fc_die("ERROR: csize %lld out of bounds (max %lld)", csize,
compr_size);
    if (fread(compr, 1, (size_t) csize, fi) != (size_t) csize) ncl6fc_die("ERROR: stream ended while reading compressed
image");
    for (int y = 0; y < benchmark_loops; y++)

```

```

        nbytes = ncl6fc_decompress(config, img, imgsize, compr, csize);
        if (nbytes < 0) ncl6fc_die("%s", ncl6fc_get_error(config));
        if (nbytes != csize) ncl6fc_die("ERROR: ncl6fc_decompress returned a wrong number of bytes %lld %lld", nbytes,
csize);
        if (fwrite(img, 1, (size_t) imgsize, fo) != (size_t) imgsize) ncl6fc_die("ERROR: could not write the decompressed
buffer");
        dn += imgsize;
        cn += csize;
        fprintf(stderr, "uncompressed: %10lld    compressed: %10lld    ratio: %.4f\n", imgsize, csize, imgsize > 0 ?
(double)csize/(double)imgsize : 0.0);
    }
    fclose(fi);
    if (fclose(fo) != 0) ncl6fc_die("ERROR: could not finish writing '%s': %s", file_out, strerror(errno));
    double dt = ncl6fc_toc(t0);
    fprintf(stderr, "SUMMARY: uncompressed: %10lld    compressed: %10lld    ratio: %.4f    speed: %.0f MiB/s\n", dn, cn, dn >
0 ? (double)cn/(double)dn : 0.0, dt > 0 ? (double)benchmark_loops*(double)dn/dt/1048576.0 : 0.0);
    free(compr);
    free(img);
    ncl6fc_deinit(config);
    free(memory);
    return 0;
}

```